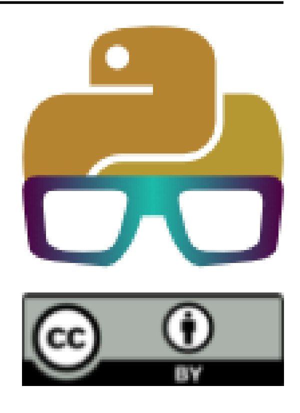



# Aprende Python

Sergio Delgado Quintero

05 de junio de 2023

# Core

1 Introducción 3
1.1 Hablando con la máquina . . . 3
1.2 Algo de historia . . . 7
1.3 Python . . . 13

2 Entornos de desarrollo 23
2.1 Thonny . . . 24
2.2 Contexto real . . . 28
2.3 VSCode . . . 37

3 Tipos de datos 47
3.1 Datos . . . 48
3.2 Números . . . 60
3.3 Cadenas de texto . . . 75

4 Control de flujo 101
4.1 Condicionales . . . 102
4.2 Bucles . . . 125

5 Estructuras de datos 143
5.1 Listas . . . 144
5.2 Tuplas . . . 175
5.3 Diccionarios . . . 182
5.4 Conjuntos . . . 200
5.5 Ficheros . . . 209

6 Modularidad 219
6.1 Funciones . . . 220
6.2 Objetos y Clases . . . 268
6.3 Excepciones . . . 322

6.4 Módulos . . . . . . . . . . 334

**7 Procesamiento de texto** **345**
7.1 re . . . . . . . . . . 346
7.2 string . . . . . . . . . . 357

**8 Acceso a datos** **361**
8.1 sqlite . . . . . . . . . . 362

**9 Ciencia de datos** **383**
9.1 jupyter . . . . . . . . . . 384
9.2 numpy . . . . . . . . . . 408
9.3 pandas . . . . . . . . . . 451
9.4 matplotlib . . . . . . . . . . 515

**10 Scraping** **557**
10.1 requests . . . . . . . . . . 558
10.2 beautifulsoup . . . . . . . . . . 566
10.3 selenium . . . . . . . . . . 581



Curso gratuito para aprender el lenguaje de programación **Python** con un enfoque **práctico**, incluyendo **ejercicios** y cobertura para distintos **niveles de conocimiento**.[1]

Este proyecto va de la mano con pycheck una herramienta que permite **trabajar todos los ejercicios propuestos** con casos de prueba incluidos y verificación de los resultados.



---

**Consejo:** **«Programming is not about typing, it's about thinking.»** – Rich Hickey

---

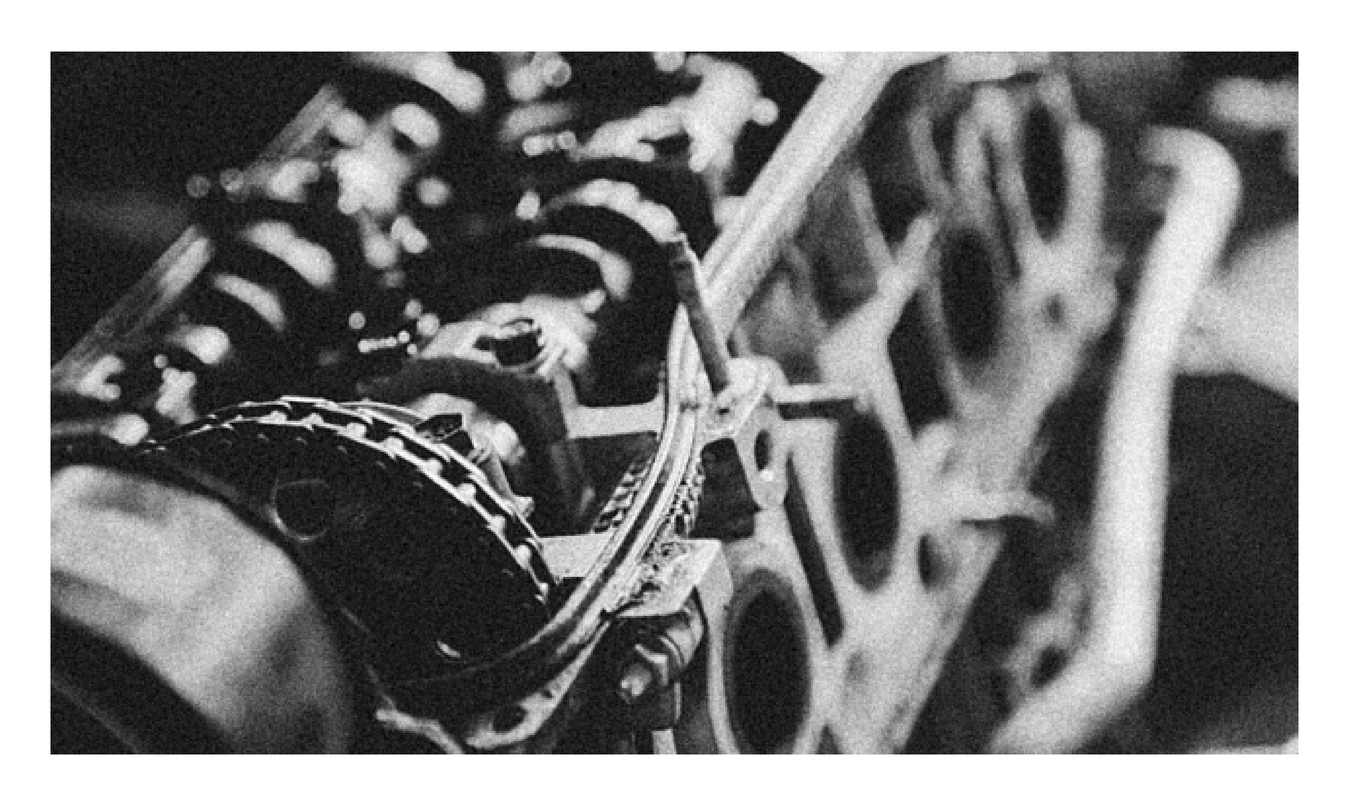
[1] En la foto de portada aparecen los Monty Python. Fuente: noticiascyl

# CAPÍTULO 1

# Introducción

Este capítulo es una introducción a la programación para conocer, desde un enfoque sencillo pero aclaratorio, los mecanismos que hay detrás de ello.

## 1.1 Hablando con la máquina



Los ordenadores son dispositivos complejos pero están diseñados para hacer una cosa bien: **ejecutar aquello que se les indica**. La cuestión es cómo indicar a un ordenador lo que queremos que ejecute. Esas indicaciones se llaman técnicamente **instrucciones** y se expresan en un lenguaje. Podríamos decir que **programar** consiste en escribir instrucciones para que sean ejecutadas por un ordenador. El lenguaje que utilizamos para ello se denomina **lenguaje de programación**.[1]

### 1.1.1 Código máquina

Pero aún seguimos con el problema de cómo hacer que un ordenador (o máquina) entienda el lenguaje de programación. A priori podríamos decir que un ordenador sólo entiende un lenguaje muy «simple» denominado código máquina. En este lenguaje se utilizan únicamente los símbolos **0** y **1** en representación de los *niveles de tensión* alto y bajo, que al fin y al cabo, son los estados que puede manejar un circuito digital. Hablamos de sistema binario. Si tuviéramos que escribir programas de ordenador en este formato sería una tarea ardua, pero afortunadamente se han ido creando con el tiempo lenguajes de programación intermedios que, posteriormente, son convertidos a código máquina.

Si intentamos visualizar un programa en código máquina, únicamente obtendríamos una secuencia de ceros y unos:

```
00001000 00000010 01111011 10101100 10010111 11011001 01000000 01100010
00110100 00010111 01101111 10111001 01010110 00110001 00101010 00011111
10000011 11001101 11110101 01001110 01010010 10100001 01101010 00001111
11101010 00100111 11000100 01110101 11011011 00010110 10011111 01010110
```

### 1.1.2 Ensamblador

El primer lenguaje de programación que encontramos en esta «escalada» es **ensamblador**. Veamos un ejemplo de código en ensamblador del típico programa que se escribe por primera vez, el *«Hello, World»*:

```
SYS_SALIDA equ 1

section .data
    msg db "Hello, World",0x0a
    len equ $ - msg ;longitud de msg

section .text
global _start ;para el linker
_start: ;marca la entrada
    mov eax, 4 ;llamada al sistema (sys_write)
```

(continúe en la próxima página)

[1] Foto original por Garett Mizunaka en Unsplash.

(proviene de la página anterior)

```
    mov ebx, 1 ;descripción de archivo (stdout)
    mov ecx, msg ;msg a escribir
    mov edx, len ;longitud del mensaje
    int 0x80 ;llama al sistema de interrupciones

fin: mov eax, SYS_SALIDA ;llamada al sistema (sys_exit)
    int 0x80
```

Aunque resulte difícil de creer, lo «único» que hace este programa es mostrar en la pantalla de nuestro ordenador la frase «Hello, World», pero además teniendo en cuenta que sólo funcionará para una arquitectura x86.

### 1.1.3 C

Aunque el lenguaje ensamblador nos facilita un poco la tarea de desarrollar programas, sigue siendo bastante complicado ya que las instrucciones son muy específicas y no proporcionan una semántica entendible. Uno de los lenguajes que vino a suplir – en parte – estos obstáculos fue C. Considerado para muchas personas como un referente en cuanto a los lenguajes de programación, permite hacer uso de instrucciones más claras y potentes. El mismo ejemplo anterior del programa *«Hello, World»* se escribiría así en lenguaje *C*:

```
#include <stdio.h>

int main() {
    printf("Hello, World");
    return 0;
}
```

### 1.1.4 Python

Si seguimos «subiendo» en esta lista de lenguajes de programación, podemos llegar hasta Python. Se dice que es un lenguaje de *más alto nivel* en el sentido de que sus instrucciones son más entendibles por un humano. Veamos cómo se escribiría el programa *«Hello, World»* en el lenguaje de programación Python:

```
print('Hello, World')
```

¡Pues así de fácil! Hemos pasado de *código máquina* (ceros y unos) a *código Python* en el que se puede entender perfectamente lo que estamos indicando al ordenador. La pregunta que surge es: ¿cómo entiende una máquina lo que tiene que hacer si le pasamos un programa hecho en Python (o cualquier otro lenguaje de alto nivel)? La respuesta es un **compilador**.

### 1.1.5 Compiladores

Los compiladores son programas que convierten un lenguaje «cualquiera» en *código máquina*. Se pueden ver como traductores, permitiendo a la máquina interpretar lo que queremos hacer.

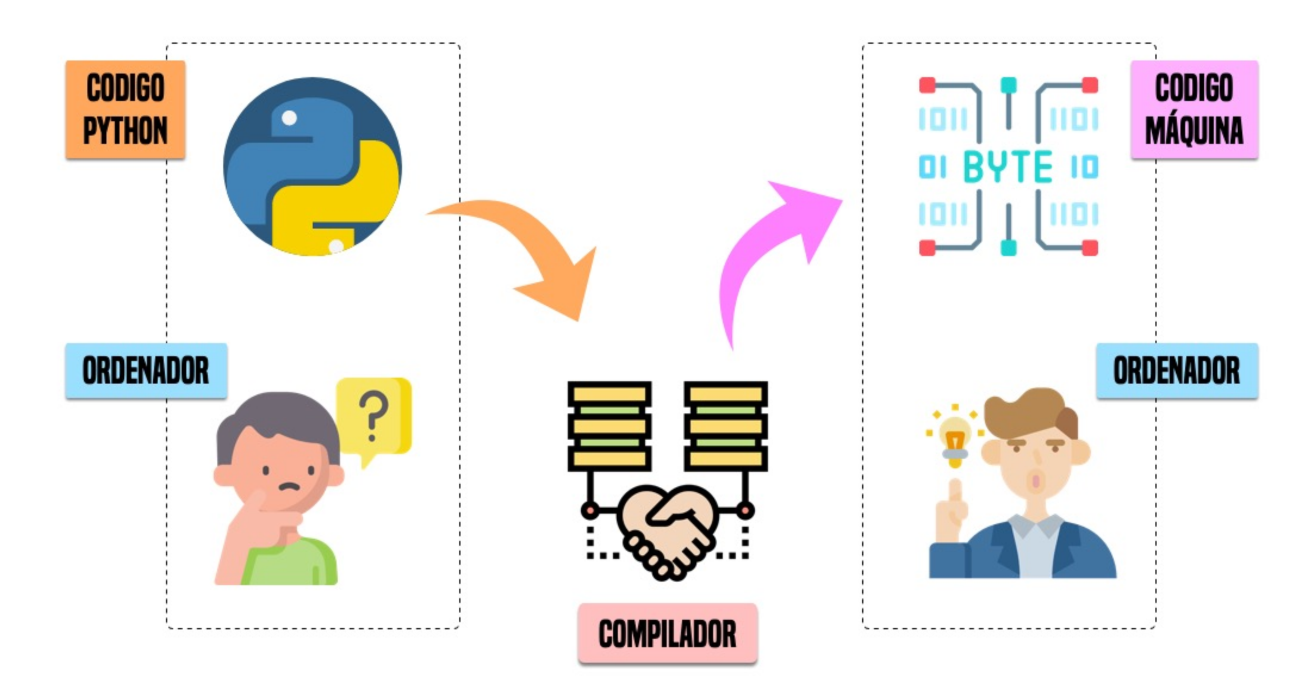



Figura 1: Esquema de funcionamiento de un compilador[2]

En el caso particular de Python el proceso de compilación genera un código intermedio denominado **bytecode**.

Si partimos del ejemplo anterior:

```
print('Hello, World')
```

el programa se compilaría[3] al siguiente «bytecode»:

```
0           0 RESUME                   0

1           2 PUSH_NULL
            4 LOAD_NAME                0 (print)
            6 LOAD_CONST               0 ('Hello, World')
            8 PRECALL                  1
           12 CALL                     1
           22 RETURN_VALUE
```

[2] Iconos originales por Flaticon.
[3] Véase más información sobre el intérprete de bytecode.

A continuación estas instrucciones básicas son ejecutadas por el intérprete de «bytecode» de Python (o máquina virtual).

**Nota:** Si queremos ver una diferencia entre un lenguaje compilado como C y un lenguaje «interpretado» como Python es que, aunque ambos realizan un proceso de traducción del código fuente, la compilación de C genera un código objeto que debe ser ejecutado en una segunda fase explícita, mientras que la compilación de Python genera un «bytecode» que se ejecuta (interpreta) de forma aparentemente simultánea.

## 1.2 Algo de historia



La historia de la programación está relacionada directamente con la aparición de los computadores, que ya desde el siglo XV tuvo sus inicios con la construcción de una máquina que realizaba operaciones básicas y raíces cuadradas (Gottfried Wilheml von Leibniz); aunque en realidad la primera gran influencia hacia la creación de los computadores fue la máquina diferencial para el cálculo de polinomios, proyecto no concluido de Charles Babbage (1793-1871) con el apoyo de Lady Ada Countess of Lovelace (1815-1852), primera persona que incursionó en la programación y de quien proviene el nombre del lenguaje de programación ADA creado por el DoD (Departamento de defensa de Estados Unidos) en la década de 1970.[1]

[1] Foto original por Dario Veronesi en Unsplash.

### 1.2.1 Hitos de la computación

La siguiente tabla es un resumen de los principales hitos en la historia de la computación:

Tabla 1: Hitos en la computación

| Personaje | Aporte | Año |
|---|---|---|
| Gottfried Leibniz | Máquinas de **operaciones básicas** | XV |
| Charles Babbage | **Máquina diferencial** para el cálculo de polinomios | XVII |
| Ada Lovelace | Matemática, informática y escritora británica. **Primera programadora** de la historia por el desarrollo de algoritmos para la máquina analítica de Babbage | XVII |
| George Boole | Contribuyó al **algebra binaria** y a los sistemas de circuitos de computadora (álgebra booleana) | 1854 |
| Herman Hollerit | Creador de un **sistema para automatizar** la pesada tarea del censo | 1890 |
| Alan Turing | **Máquina de Turing** - una máquina capaz de resolver problemas - Aportes de Lógica Matemática - Computadora con tubos de vacío | 1936 |
| John Atanasoff | Primera **computadora digital** electrónica patentada: Atanasoff Berry Computer (ABC) | 1942 |
| Howard Aiken | En colaboración con IBM desarrolló el **Mark I**, una computadora electromecánica de 16 metros de largo y más de dos de alto que podía realizar las cuatro operaciones básicas y trabajar con información almacenada en forma de tablas | 1944 |
| Grace Hopper | Primera **programadora** que utilizó el Mark I | 1945 |
| John W. Mauchly | Junto a John Presper Eckert desarrolló una **computadora electrónica** completamente operacional a gran escala llamada Electronic Numerical Integrator And Computer (ENIAC) | 1946 |
| John Von Neumann | Propuso guardar en **memoria** no solo la **información**, sino también los **programas**, acelerando los procesos | 1946 |

Luego los avances en las ciencias informáticas han sido muy acelerados, se reemplazaron los tubos de vacío por transistores en 1958 y en el mismo año, se sustituyeron por circuitos integrados, y en 1961 se miniaturizaron en **chips de silicio**. En 1971 apareció el primer microprocesador de Intel; y en 1973 el primer sistema operativo CP/M. El primer computador personal es comercializado por IBM en el año 1980.

[2] Fuente: Meatze.



Figura 2: Ada Lovelace: primera programadora de la historia[2]

### 1.2.2 De los computadores a la programación

De acuerdo a este breve viaje por la historia, la programación está vinculada a la aparición de los computadores, y los lenguajes tuvieron también su evolución. Inicialmente, como ya hemos visto, se programaba en **código binario**, es decir en cadenas de 0s y 1s, que es el lenguaje que entiende directamente el computador, tarea extremadamente difícil; luego se creó el **lenguaje ensamblador**, que aunque era lo mismo que programar en binario, al estar en letras era más fácil de recordar. Posteriormente aparecieron **lenguajes de alto nivel**, que en general, utilizan palabras en inglés, para dar las órdenes a seguir, para lo cual utilizan un proceso intermedio entre el lenguaje máquina y el nuevo código llamado código fuente, este proceso puede ser un compilador o un intérprete.

Un **compilador** lee todas las instrucciones y genera un resultado; un **intérprete** ejecuta y genera resultados línea a línea. En cualquier caso han aparecido nuevos lenguajes de programación, unos denominados estructurados y en la actualidad en cambio los lenguajes orientados a objetos y los lenguajes orientados a eventos.[3]

### 1.2.3 Cronología de lenguajes de programación

Desde la década de 1950 se han sucedido multitud de lenguajes de programación que cada vez incorporan más funcionalidades destinadas a cubrir las necesidades del desarrollo de aplicaciones. A continuación se muestra una tabla con la historia de los lenguajes de programación más destacados:

El **número** actual de lenguajes de programación depende de lo que se considere un *lenguaje de programación* y a *quién se pregunte*. Según TIOBE más de 250; según Wikipedia más de 700, según Language List más de 2500; y para una cifra muy alta podemos considerar a Online Historical Encyclopaedia of Programming Languages que se acerca a los **9000**.

### 1.2.4 Creadores de lenguajes de programación

El avance de la computación está íntimamente relacionado con el desarrollo de los lenguajes de programación. Sus creadores y creadoras juegan un *rol fundamental* en la historia tecnológica. Veamos algunas de estas personas:[4]

Tabla 2: Creadores de lenguajes de programación

| Personaje | Aporte |
|---|---|
| Alan Cooper | Desarrollador de **Visual Basic** |
| Alan Kay | Pionero en programación orientada a objetos. Creador de **Smalltalk** |

continué en la próxima página

[3] Fuente: Universidad Técnica del Norte.
[4] Fuente: Wikipedia.

Tabla 2 – proviene de la página anterior

| Personaje | Aporte |
|---|---|
| Anders Hejlsberg | Desarrollador de **Turbo Pascal**, **Delphi** y **C#** |
| Bertrand Meyer | Inventor de **Eiffel** |
| Bill Joy | Inventor de **vi**. Autor de **BSD Unix**. Creador de **SunOS**, el cual se convirtió en **Solaris** |
| Bjarne Stroustrup | Desarrollador de **C++** |
| Dennis Ritchie | Inventor del **lenguaje C** y del Sistema Operativo **Unix** |
| Brian Kernighan | Coautor del primer libro de programación en lenguaje **C** con Dennis Ritchie y coautor de los lenguajes de programación **AWK** y **AMPL** |
| Edsger W. Dijkstra | Desarrolló las bases para la programación estructurada. Algoritmo de caminos mínimos |
| Grace Hopper | Desarrolladora de **Flow-Matic**, influenciando el lenguaje **COBOL** |
| Guido van Rossum | Creador de **Python** |
| James Gosling | Desarrollador de **Oak**. Precursor de **Java** |
| Joe Armstrong | Creador de **Erlang** |
| John Backus | Inventor de **Fortran** |
| John McCarthy | Inventor de **LISP** |
| John von Neumann | Creador del concepto de sistema operativo |
| Ken Thompson | Inventor de B. Desarrollador de **Go**. Coautor del sistema operativo Unix |
| Kenneth E. Iverson | Desarrollador de **APL**. Co-desarrollador de **J** junto a Roger Hui |
| Larry Wall | Creador de **Perl** y Perl 6 |
| Martin Odersky | Creador de **Scala**. Previamente contribuyó en el diseño de Java |
| Mitchel Resnick | Creador del lenguaje visual **Scratch** |
| Nathaniel Rochester | Inventor del primer lenguaje en **ensamblador** simbólico (IBM 701) |
| Niklaus Wirth | Inventor de **Pascal**, **Modula** y **Oberon** |
| Robin Milner | Inventor de **ML**. Compartió crédito en el método Hindley–Milner de inferencia de tipo polimórfica |
| Seymour Papert | Pionero de la inteligencia artificial. Inventor del lenguaje de programación **Logo** en 1968 |
| Stephen Wolfram | Creador de **Mathematica** |
| Yukihiro Matsumoto | Creador de **Ruby** |

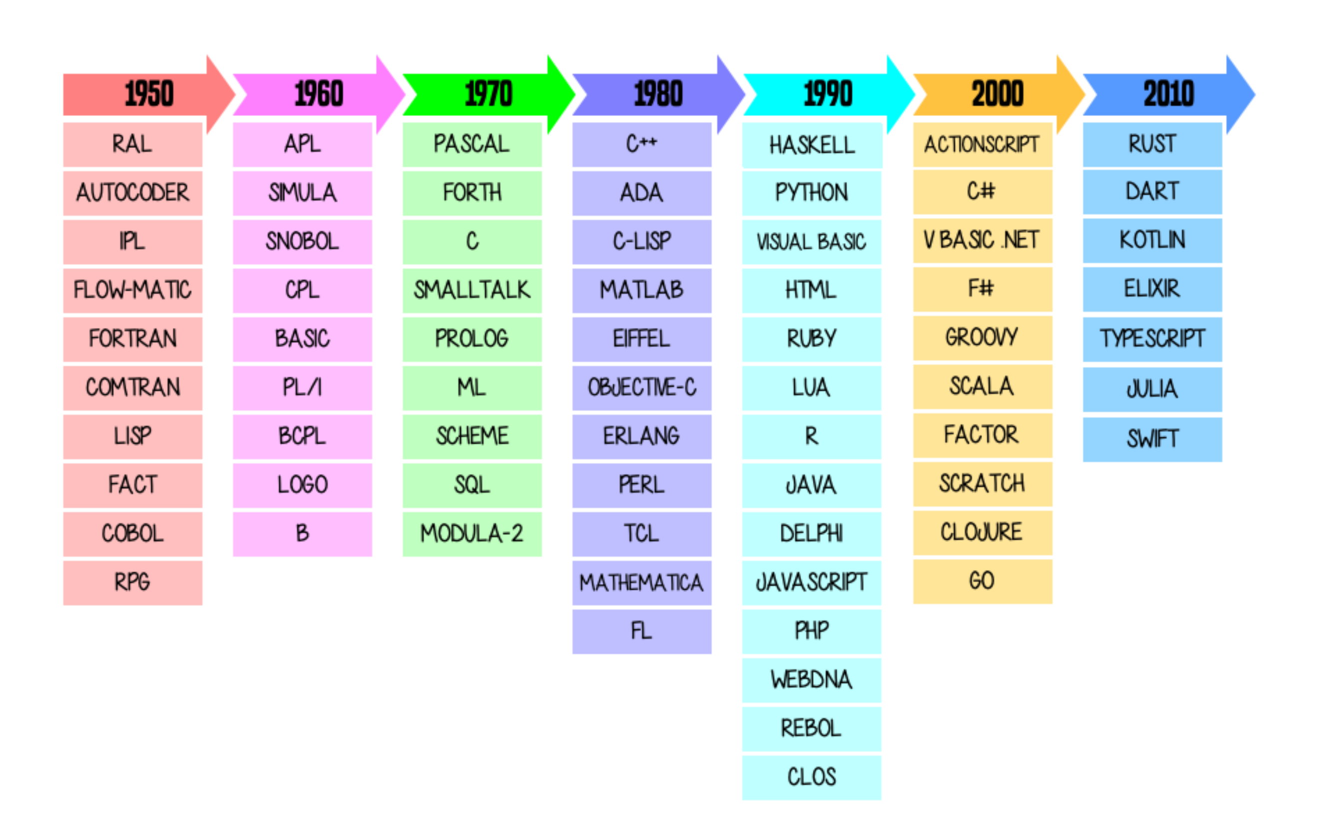



Figura 3: Cronología de los lenguajes de programación más destacados

## 1.3 Python



Python es un lenguaje de programación de *alto nivel* creado a finales de los 80/principios de los 90 por Guido van Rossum, holandés que trabajaba por aquella época en el *Centro para las Matemáticas y la Informática* de los Países Bajos. Sus instrucciones están muy cercanas al **lenguaje natural** en inglés y se hace hincapié en la **legibilidad** del código. Toma su nombre de los Monty Python, grupo humorista de los 60 que gustaban mucho a Guido. Python fue creado como sucesor del lenguaje `ABC`.[1]

### 1.3.1 Características del lenguaje

A partir de su definición de la Wikipedia:

- Python es un lenguaje de programación **interpretado** y **multiplataforma** cuya filosofía hace hincapié en una sintaxis que favorezca un **código legible**.
- Se trata de un lenguaje de programación **multiparadigma**, ya que soporta **orientación a objetos, programación imperativa** y, en menor medida, programación funcional.
- Añadiría, como característica destacada, que se trata de un lenguaje de **propósito general**.

[1] Foto original por Markéta Marcellová en Unsplash.

### Ventajas

- Libre y gratuito (OpenSource).
- Fácil de leer, parecido a pseudocódigo.
- Aprendizaje relativamente fácil y rápido: claro, intuitivo….
- Alto nivel.
- Alta Productividad: simple y rápido.
- Tiende a producir un buen código: orden, limpieza, elegancia, flexibilidad, …
- Multiplataforma. Portable.
- Multiparadigma: programación imperativa, orientada a objetos, funcional, …
- Interactivo, modular, dinámico.
- Librerías extensivas («pilas incluídas»).
- Gran cantidad de librerías de terceros.
- Extensible (C++, C, …) y «embebible».
- Gran comunidad, amplio soporte.
- Interpretado.
- Tipado dinámico[5].
- Fuertemente tipado[6].

### Desventajas

- Interpretado (velocidad de ejecución, multithread vs GIL, etc.).
- Consumo de memoria.
- Errores no detectables en tiempo de compilación.
- Desarrollo móvil.
- Documentación a veces dispersa e incompleta.
- Varios módulos para la misma funcionalidad.
- Librerías de terceros no siempre del todo maduras.

[5] Tipado dinámico significa que una variable puede cambiar de tipo durante el tiempo de vida de un programa. C es un lenguaje de tipado estático.

[6] Fuertemente tipado significa que, de manera nativa, no podemos operar con dos variables de tipos distintos, a menos que realice una conversión explícita. Javascript es un lenguaje débilmente tipado.

### 1.3.2 Uso de Python

Al ser un lenguaje de propósito general, podemos encontrar aplicaciones prácticamente en todos los campos científico-tecnológicos:

- Análisis de datos.
- Aplicaciones de escritorio.
- Bases de datos relacionales / NoSQL
- Buenas prácticas de programación / Patrones de diseño.
- Concurrencia.
- Criptomonedas / Blockchain.
- Desarrollo de aplicaciones multimedia.
- Desarrollo de juegos.
- Desarrollo en dispositivos embebidos.
- Desarrollo web.
- DevOps / Administración de sistemas / Scripts de automatización.
- Gráficos por ordenador.
- Inteligencia artificial.
- Internet de las cosas.
- Machine Learning.
- Programación de parsers / scrapers / crawlers.
- Programación de redes.
- Propósitos educativos.
- Prototipado de software.
- Seguridad.
- Tests automatizados.

De igual modo son muchas las empresas, instituciones y organismos que utilizan Python en su día a día para mejorar sus sistemas de información. Veamos algunas de las más relevantes:

Existen ránkings y estudios de mercado que sitúan a Python como uno de los lenguajes más *usados* y la vez, más *amados* dentro del mundo del desarrollo de software.

En el momento de la escritura de este documento, la última actualización del Índice TIOBE es de *marzo de 2023* en el que Python ocupaba el **primer lugar de los lenguajes de programación más usados**, por delante de *C* y *Java*.




Figura 4: Grandes empresas y organismos que usan Python

| Mar 2023 | Mar 2022 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Python | 14.83% | +0.57% |
| 2 | 2 | | C | 14.73% | +1.67% |
| 3 | 3 | | Java | 13.56% | +2.37% |

Figura 5: Índice TIOBE 2023

Igualmente en la encuesta a desarrolladores de Stack Overflow *hecha en 2022*, Python ocupaba el **cuarto puesto de los lenguajes de programación más populares**, sólo por detrás de *Javascript*, *HTML/CSS* y *SQL*:




Figura 6: Encuesta Stack Overflow 2022

### 1.3.3 Versiones de Python

En el momento de la escritura de este material, se muestra a continuación la evolución de las versiones mayores de Python a lo largo de la historia:[3]

| Versión | Fecha de lanzamiento |
|---|---|
| Python 1.0 | Enero 1994 |
| Python 1.5 | Diciembre 1997 |
| Python 1.6 | Septiembre 2000 |
| Python 2.0 | Octubre 2000 |
| Python 2.1 | Abril 2001 |
| Python 2.2 | Diciembre 2001 |
| Python 2.3 | Julio 2003 |
| Python 2.4 | Noviembre 2004 |
| Python 2.5 | Septiembre 2006 |
| Python 2.6 | Octubre 2008 |
| Python 2.7 | Julio 2010 |
| **Python 3.0** | **Diciembre 2008** |
| Python 3.1 | Junio 2009 |
| Python 3.2 | Febrero 2011 |
| Python 3.3 | Septiembre 2012 |
| Python 3.4 | Marzo 2014 |
| Python 3.5 | Septiembre 2015 |

continué en la próxima página

[3] Fuente: python.org.

Tabla 3 – proviene de la página anterior

| Versión | Fecha de lanzamiento |
|---|---|
| Python 3.6 | Diciembre 2016 |
| Python 3.7 | Junio 2018 |
| Python 3.8 | Octubre 2019 |
| Python 3.9 | Octubre 2020 |
| Python 3.10 | Octubre 2021 |
| Python 3.11 | Octubre 2022 |
| *Python 3.12* | *Octubre 2023* ? |

Un dato curioso, o directamente un «frikismo»: Desde Python 3.8, cada nueva versión estable sale a la luz en el mes de **Octubre**. En este escenario de Python *3.version* se cumplen las siguientes igualdades:

$$version = year - 2011$$
$$year = version + 2011$$

El cambio de **Python 2** a **Python 3** fue bastante «traumático» ya que se **perdió la compatibilidad** en muchas de las estructuras del lenguaje. Los «*core-developers*»[4], con *Guido van Rossum* a la cabeza, vieron la necesidad de aplicar estas modificaciones en beneficio del rendimiento y expresividad del lenguaje de programación. Este cambio implicaba que el código escrito en Python 2 no funcionaría (de manera inmediata) en Python 3.

El pasado **1 de enero de 2020** finalizó oficialmente el **soporte a la versión 2.7** del lenguaje de programación Python. Es por ello que se recomienda lo siguiente:

- Si aún desarrollas aplicaciones escritas en Python 2, deberías migrar a Python 3.
- Si vas a desarrollar una nueva aplicación, deberías hacerlo directamente en Python 3.

**Importante:** Únete a **Python 3** y aprovecha todas sus ventajas.

### 1.3.4 CPython

**Nivel avanzado**

Existen múltiples **implementaciones** de Python según el lenguaje de programación que se ha usado para desarrollarlo. Veamos algunas de ellas:

[4] Término que se refiere a los/las desarrolladores/as principales del lenguaje de programación.

| Implementación | Lenguaje |
|---|---|
| CPython | C |
| Jython | Java |
| IronPython | C# |
| Brython | JavaScript |
| RustPython | Rust |
| MicroPython | C |

**Nota:** Cuando hacemos referencia a Python hablamos (implícitamente) de CPython. Este manual versa exclusivamente sobre CPython.

### 1.3.5 Zen de Python

Existen una serie de *reglas* «filosóficas» que indican una manera de hacer y de pensar dentro del mundo **pitónico**[2] creadas por Tim Peters, llamadas el Zen de Python y que se pueden aplicar incluso más allá de la programación:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

En su traducción de la Wikipedia:

[2] Dícese de algo/alguien que sigue las convenciones de Python.

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

**Ver también:**

Si quieres darle un toque a tu escritorio, puedes descargar este fondo de pantalla del Zen de Python que queda muy chulo.

### 1.3.6 Consejos para programar

Un listado de consejos muy interesantes cuando nos enfrentamos a la programación, basados en la experiencia de @codewithvoid:

1. Escribir código es el último paso del proceso.
2. Para resolver problemas: pizarra mejor que teclado.
3. Escribir código sin planificar = estrés.
4. Pareces más inteligente siendo claro, no siendo listo.

5. La constancia a largo plazo es mejor que la intensidad a corto plazo.
6. La solución primero. La optimización después.
7. Gran parte de la programación es resolución de problemas.
8. Piensa en múltiples soluciones antes de decidirte por una.
9. Se aprende construyendo proyectos, no tomando cursos.
10. Siempre elije simplicidad. Las soluciones simples son más fáciles de escribir.
11. Los errores son inevitables al escribir código. Sólo te informan sobre lo que no debes hacer.
12. Fallar es barato en programación. Aprende mediante la práctica.
13. Gran parte de la programación es investigación.
14. La programación en pareja te enseñará mucho más que escribir código tu solo.
15. Da un paseo cuando estés bloqueado con un error.
16. Convierte en un hábito el hecho de pedir ayuda. Pierdes cero credibilidad pidiendo ayuda.
17. El tiempo gastado en entender el problema está bien invertido.
18. Cuando estés bloqueado con un problema: sé curioso, no te frustres.
19. Piensa en posibles escenarios y situaciones extremas antes de resolver el problema.
20. No te estreses con la sintaxis de lenguaje de programación. Entiende conceptos.
21. Aprende a ser un buen corrector de errores. Esto se amortiza.
22. Conoce pronto los atajos de teclado de tu editor favorito.
23. Tu código será tan claro como lo tengas en tu cabeza.
24. Gastarás el doble de tiempo en corregir errores que en escribir código.
25. Saber buscar bien en Google es una habilidad valiosa.
26. Lee código de otras personas para inspirarte.
27. Únete a comunidades de desarrollo para aprender con otros/as programadores/as.

CAPÍTULO 2

# Entornos de desarrollo

Para poder utilizar Python debemos preparar nuestra máquina con las herramientas necesarias. Este capítulo trata sobre la instalación y configuración de los elementos adecuados para el desarrollo con el lenguaje de programación Python.

## 2.1 Thonny



Thonny es un programa muy interesante para empezar a aprender Python, ya que engloba tres de las herramientas fundamentales para trabajar con el lenguaje: **intérprete**, **editor** y **depurador**.[1]

Cuando vamos a trabajar con Python debemos tener instalado, como mínimo, un *intérprete* del lenguaje (para otros lenguajes sería un *compilador*). El **intérprete** nos permitirá *ejecutar* nuestro código para obtener los resultados deseados. La idea del intéprete es lanzar instrucciones «sueltas» para probar determinados aspectos.

Pero normalmente queremos ir un poco más allá y poder escribir programas algo más largos, por lo que también necesitaremos un **editor**. Un editor es un programa que nos permite crear ficheros de código (en nuestro caso con extensión `*.py`), que luego son ejecutados por el intérprete.

Hay otra herramienta interesante dentro del entorno de desarrollo que sería el **depurador**. Lo podemos encontrar habitualmente en la bibliografía por su nombre inglés *debugger*. Es el módulo que nos permite ejecutar paso a paso nuestro código y visualizar qué está ocurriendo en cada momento. Se suele usar normalmente para encontrar fallos (*bugs*) en nuestros programas y poder solucionarlos (*debug*/*fix*).

Cuando nos encontramos con un programa que proporciona estas funciones (e incluso otras adicionales) para el trabajo de programación, nos referimos a él como un *Entorno Integrado de Desarrollo*, conocido popularmente por sus siglas en inglés **IDE** (por Integrated Development

[1] Foto original de portada por freddie marriage en Unsplash.

Environment). Thonny es un IDE gratuito, sencillo y apto para principiantes.

### 2.1.1 Instalación

Para instalar Thonny debemos acceder a su web y descargar la aplicación para nuestro sistema operativo. La ventaja es que está disponible tanto para **Windows**, **Mac** y **Linux**. Una vez descargado el fichero lo ejecutamos y seguimos su instalación paso por paso.

Una vez terminada la instalación ya podemos lanzar la aplicación que se verá parecida a la siguiente imagen:

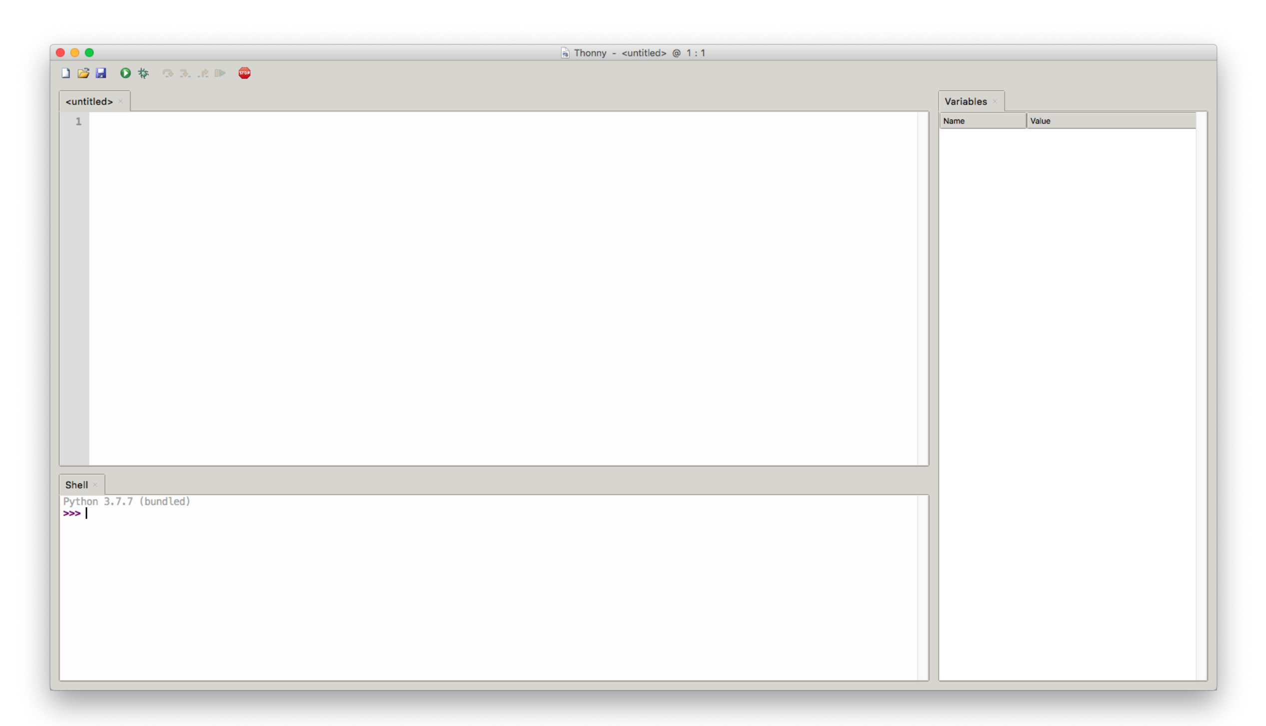



Figura 1: Aspecto de Thonny al arrancarlo

---

**Nota:** Es posible que el aspecto del programa varíe ligeramente según el sistema operativo, configuración de escritorio, versión utilizada o idioma (*en mi caso está en inglés*), pero a efectos de funcionamiento no hay diferencia.

---

Podemos observar que la pantalla está dividida en 3 paneles:

- *Panel principal* que contiene el **editor** e incluye la etiqueta `<untitled>` donde escribiremos nuestro *código fuente* Python.
- *Panel inferior* con la etiqueta *Shell* que contiene el **intérprete** de Python. En el momento de la escritura del presente documento, Thonny incluye la versión de Python 3.7.7.

- *Panel derecho* que contiene el **depurador**. Más concretamente se trata de la ventana de variables donde podemos *inspeccionar* el valor de las mismas.

### 2.1.2 Probando el intérprete

El intérprete de Python (por lo general) se identifica claramente porque posee un **prompt**[2] con tres angulos hacia la derecha `>>>`. En Thonny lo podemos encontrar en el panel inferior, pero se debe tener en cuenta que el intérprete de Python es una herramienta autocontenida y que la podemos ejecutar desde el símbolo del sistema o la terminal:

Lista 1: Invocando el intérprete de Python 3.7 desde una terminal en MacOS

```
$ python3.7
Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Para hacer una prueba inicial del intérprete vamos a retomar el primer programa que se suele hacer. Es el llamado *«Hello, World»*. Para ello escribimos lo siguiente en el intérprete y pulsamos la tecla `ENTER`:

```
>>> print('Hello, World')
Hello, World
```

Lo que hemos hecho es indicarle a Python que ejecute como **entrada** la instrucción `print('Hello, World')`. La **salida** es el texto `Hello, World` que lo vemos en la siguiente línea (*ya sin el prompt* `>>>`).

### 2.1.3 Probando el editor

Ahora vamos a realizar la misma operación, pero en vez de ejecutar la instrucción directamente en el intérprete, vamos a crear un fichero y guardarlo con la sentencia que nos interesa. Para ello escribimos `print('Hello, World')` en el panel de edición (*superior*) y luego guardamos el archivo con el nombre `helloworld.py`[3]:

---

**Importante:** Los ficheros que contienen programas hechos en Python siempre deben tener la extensión `.py`

---

[2] Término inglés que se refiere al símbolo que precede la línea de comandos.

[3] La carpeta donde se guarden los archivos de código no es crítico para su ejecución, pero sí es importante mantener un orden y una organización para tener localizados nuestros ficheros y proyectos.

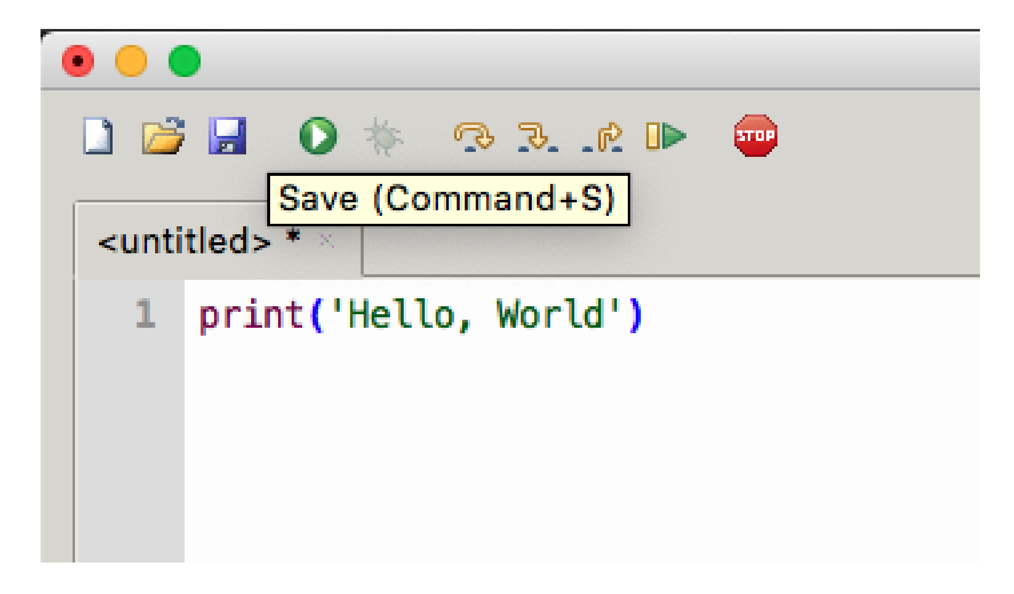



Figura 2: Guardando nuestro primer programa en Python

Ahora ya podemos *ejecutar* nuestro fichero `helloworld.py`. Para ello pulsamos el botón verde con triángulo blanco (en la barra de herramientas) o bien damos a la tecla `F5`. Veremos que en el panel de *Shell* nos aparece la salida esperada. Lo que está pasando «entre bambalinas» es que el intérprete de Python está recibiendo como entrada el fichero que hemos creado; lo ejecuta y devuelve la salida para que Thonny nos lo muestre en el panel correspondiente.

### 2.1.4 Probando el depurador

Nos falta por probar el depurador o «debugger». Aunque su funcionamiento va mucho más allá, de momento nos vamos a quedar en la posibilidad de inspeccionar las variables de nuestro programa. Desafortunadamente `helloworld.py` es muy simple y ni siquiera contiene variables, pero podemos hacer una pequeña modificación al programa para poder incorporarlas:

```
1 msg = 'Hello, World'
2 print(msg)
```

Aunque ya lo veremos en profundidad, lo que hemos hecho es añadir una variable `msg` en la *línea 1* para luego utilizarla al mostrar por pantalla su contenido. Si ahora volvemos a ejecutar nuestro programa veremos que en el panel de variables nos aparece la siguiente información:

| Name | Value |
|---|---|
| msg | 'Hello, World' |

También existe la posibilidad, a través del depurador, de ir ejecutando nuestro programa **paso a paso**. Para ello basta con pulsar en el botón que tiene un *insecto*. Ahí comienza la

sesión de depuración y podemos avanzar instrucción por instrucción usando la tecla `F7`:

Figura 3: Depurando nuestro primer programa en Python

## 2.2 Contexto real



Hemos visto que *Thonny* es una herramienta especialmente diseñada para el aprendizaje de Python, integrando diferentes módulos que facilitan su gestión. Si bien lo podemos utilizar

para un desarrollo más *«serio»*, se suele recurrir a un flujo de trabajo algo diferente en **contextos más reales**.[1]

## 2.2.1 Python

La forma más habitual de instalar Python (junto con sus librerías) es descargarlo e instalarlo desde su página oficial:

- Versiones de Python para Windows
- Versiones de Python para Mac
- Versiones de Python para Linux

**Truco:** Tutorial para instalar Python en Windows.

### Anaconda

Otra de las alternativas para disponer de Python en nuestro sistema y que además es muy utilizada, es **Anaconda**. Se trata de un *conjunto de herramientas*, orientadas en principio a la *ciencia de datos*, pero que podemos utilizarlas para desarrollo general en Python (junto con otras librerías adicionales).

Existen versiones de pago, pero la distribución *Individual Edition* es «open-source» y gratuita. Se puede descargar desde su página web. Anaconda trae por defecto una gran cantidad de paquetes Python en su distribución.

**Ver también:**

Miniconda es un instalador mínimo que trae por defecto Python y un pequeño número de paquetes útiles.

## 2.2.2 Gestión de paquetes

La instalación limpia[2] de Python ya ofrece de por sí muchos paquetes y módulos que vienen por defecto. Es lo que se llama la librería estándar. Pero una de las características más destacables de Python es su inmenso «ecosistema» de paquetes disponibles en el Python Package Index (PyPI).

Para gestionar los paquetes que tenemos en nuestro sistema se utiliza la herramienta pip, una utilidad que también se incluye en la instalación de Python. Con ella podremos instalar,

[1] Foto original de portada por SpaceX en Unsplash.

[2] También llamada «vanilla installation» ya que es la que viene por defecto y no se hace ningúna personalización.

desinstalar y actualizar paquetes, según nuestras necesidades. A continuación se muestran las instrucciones que usaríamos para cada una de estas operaciones:

Lista 2: Instalación, desinstalación y actualización del paquete `pandas` utilizando `pip`

```
$ pip install pandas
$ pip uninstall pandas
$ pip install pandas --upgrade
```

**Consejo:** Para el caso de *Anaconda* usaríamos `conda install pandas` (aunque ya viene preinstalado).

### 2.2.3 Entornos virtuales

#### Nivel intermedio

Cuando trabajamos en distintos proyectos, no todos ellos requieren los mismos paquetes ni siquiera la misma versión de Python. La gestión de estas situaciones no es sencilla si únicamente instalamos paquetes y manejamos configuraciones a nivel global (*a nivel de máquina*). Es por ello que surge el concepto de **entornos virtuales**. Como su propio nombre indica se trata de crear distintos entornos en función de las necesidades de cada proyecto, y esto nos permite establecer qué versión de Python usaremos y qué paquetes instalaremos.

La manera más sencilla de crear un entorno virtual es la siguiente:

```
1 $ cd myproject
2 $ python -m venv --prompt myproject .venv
3 $ source .venv/bin/activate
```

- *Línea 1*: Entrar en la carpeta de nuestro proyecto.
- *Línea 2*: Crear una carpeta `.venv` con los ficheros que constituyen el entorno virtual.
- *Línea 3*: Activar el entorno virtual. A partir de aquí todo lo que se instale quedará dentro del entorno virtual.

### virtualenv

El paquete de Python que nos proporciona la funcionalidad de crear y gestionar entornos virtuales se denomina virtualenv. Su instalación es sencilla a través del gestor de paquetes `pip`:

```
$ pip install virtualenv
```

Si bien con `virtualenv` tenemos las funcionalidades necesarias para trabajar con entornos virtuales, destacaría una herramienta llamada virtualenvwrapper que funciona *por encima* de `virtualenv` y que facilita las operaciones sobre entornos virtuales. Su instalación es equivalente a cualquier otro paquete Python:

```
$ pip install virtualenvwrapper
```

Veamos a continuación algunos de los comandos que nos ofrece:

```
$ ~/project1 > mkvirtualenv env1
Using base prefix '/Library/Frameworks/Python.framework/Versions/3.7'
New python executable in /Users/sdelquin/.virtualenvs/env1/bin/python3.7
Also creating executable in /Users/sdelquin/.virtualenvs/env1/bin/python
Installing setuptools, pip, wheel...
done.
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
↪predeactivate
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
↪postdeactivate
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
↪preactivate
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
↪postactivate
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/get_
↪env_details
$ (env1) ~/project1 > pip install requests
Collecting requests
Using cached requests-2.24.0-py2.py3-none-any.whl (61 kB)
Collecting idna<3,>=2.5
Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting certifi>=2017.4.17
Using cached certifi-2020.6.20-py2.py3-none-any.whl (156 kB)
Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1
Using cached urllib3-1.25.10-py2.py3-none-any.whl (127 kB)
Collecting chardet<4,>=3.0.2
Using cached chardet-3.0.4-py2.py3-none-any.whl (133 kB)
Installing collected packages: idna, certifi, urllib3, chardet, requests
Successfully installed certifi-2020.6.20 chardet-3.0.4 idna-2.10 requests-2.24.0␣
↪urllib3-1.25.10
```

(continué en la próxima página)

(proviene de la página anterior)

```
$ (env1) ~/project1 > deactivate
$ ~/project1 > workon env1
$ (env1) ~/project1 > lssitepackages
__pycache__                 distutils-precedence.pth    pkg_resources               ␣
↪urllib3-1.25.10.dist-info
_distutils_hack             easy_install.py             requests                    ␣
↪wheel
certifi                     idna                        requests-2.24.0.dist-info   ␣
↪wheel-0.34.2.dist-info
certifi-2020.6.20.dist-info idna-2.10.dist-info         setuptools
chardet                     pip                         setuptools-49.3.2.dist-info
chardet-3.0.4.dist-info     pip-20.2.2.dist-info        urllib3
$ (env1) ~/project1 >
```

- `$ mkvirtualenv env1`: crea un entorno virtual llamado `env1`
- `$ pip install requests`: instala el paquete `requests` dentro del entorno virtual `env1`
- `$ workon env1`: activa el entorno virtual `env1`
- `$ lssitepackages`: lista los paquetes instalados en el entorno virtual activo

## pyenv

pyenv permite cambiar fácilmente entre múltiples versiones de Python en un mismo sistema. Su instalación engloba varios pasos y está bien explicada en la página del proyecto.

La mayor diferencia con respecto a *virtualenv* es que no instala las distintas versiones de Python a nivel global del sistema. En vez de eso, se suele crear una carpeta `.pyenv` en el HOME del usuario, donde todo está aislado sin generar intrusión en el sistema operativo.

Podemos hacer cosas como:

- Listar las versiones de Python instaladas:

```
$ pyenv versions
  3.7.4
  * 3.5.0 (set by /Users/yuu/.pyenv/version)
  miniconda3-3.16.0
  pypy-2.6.0
```

- Descubrir la versión global «activa» de Python:

```
$ python --version
Python 3.5.0
```

- Cambiar la versión global «activa» de Python:

```
$ pyenv global 3.7.4

$ python --version
Python 3.7.4
```

- Instalar una nueva versión de Python:

```
$ pyenv install 3.9.1
...
```

- Activar una versión de Python local por carpetas:

```
$ cd /cool-project
$ pyenv local 3.9.1
$ python --version
Python 3.9.1
```

También existe un módulo denominado pyenv-virtualenv para manejar entornos virtuales utilizando las ventajas que proporciona *pyenv*.

### 2.2.4 Editores

Existen multitud de editores en el mercado que nos pueden servir perfectamente para escribir código Python. Algunos de ellos incorporan funcionalidades extra y otros simplemente nos permiten editar ficheros. Cabe destacar aquí el concepto de **Entorno de Desarrollo Integrado**, más conocido por sus siglas en inglés **IDE**[3]. Se trata de una aplicación informática que proporciona servicios integrales para el desarrollo de software.

Podríamos decir que *Thonny* es un IDE de aprendizaje, pero existen muchos otros. Veamos un listado de editores de código que se suelen utilizar para desarrollo en Python:

- **Editores generales o IDEs con soporte para Python:**
  - Eclipse + PyDev
  - Sublime Text
  - Atom
  - GNU Emacs
  - Vi-Vim
  - Visual Studio (+ Python Tools)
  - Visual Studio Code (+ Python Tools)
- **Editores o IDEs específicos para Python:**

[3] Integrated Development Environment.

   - PyCharm
   - Spyder
   - Thonny

Cada editor tiene sus características (ventajas e inconvenientes). Supongo que la preferencia por alguno de ellos estará en base a la experiencia y a las necesidades que surjan. La parte buena es que hay diversidad de opciones para elegir.

### 2.2.5 Jupyter Notebook

Jupyter Notebook es una aplicación «open-source» que permite crear y compartir documentos que contienen código, ecuaciones, visualizaciones y texto narrativo. Podemos utilizarlo para propósito general aunque suele estar más enfocado a *ciencia de datos*: limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización o «machine-learning»[4].

Podemos verlo como un intérprete de Python (contiene un «kernel»[5] que permite ejecutar código) con la capacidad de incluir documentación en formato Markdown, lo que potencia sus funcionalidades y lo hace adecuado para preparar cualquier tipo de material vinculado con lenguajes de programación.

Aunque su uso está más extendido en el mundo Python, existen muchos otros «kernels» sobre los que trabajar en Jupyter Notebook.

**Ver también:**

Sección sobre *Jupyter*.

---

**Truco:** Visual Studio Code también dispone de integración con Jupyter Notebooks.

---

### 2.2.6 repl.it

repl.it es un **servicio web que ofrece un entorno de desarrollo integrado** para programar en más de 50 lenguajes (Python incluido).

Es gratuito y de uso colaborativo. Se requiere una cuenta en el sistema para utilizarlo. El hecho de no requerir instalación ni configuración previa lo hace atractivo en determinadas circunstancias.

En su versión gratuita ofrece:

[4] Término inglés utilizado para hacer referencia a algoritmos de aprendizaje automático.
[5] Proceso específico para un lenguaje de programación que ejecuta instrucciones y actúa como interfaz de entrada/salida.

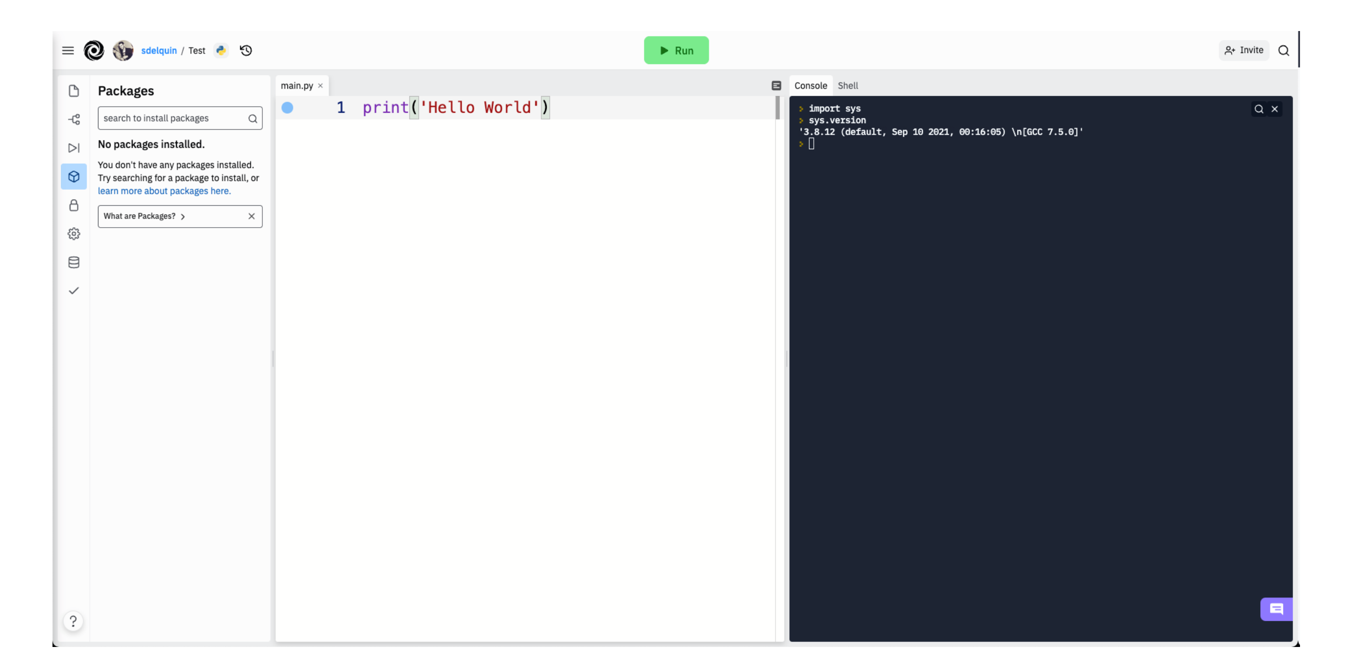



Figura 4: repl.it

- Almacenamiento de 500MB.
- Python 3.8.2 (febrero de 2022).
- 117 paquetes preinstalados (febrero de 2022).
- Navegador (y subida) de ficheros integrado.
- Gestor de paquetes integrado.
- Integración con GitHub.
- Gestión de secretos (datos sensibles).
- Base de datos clave-valor ya integrada.
- Acceso (limitado) al sistema operativo y sistema de ficheros.

### 2.2.7 WSL

Si estamos trabajando en un sistema **Windows 10** es posible que nos encontremos más cómodos usando una terminal tipo «Linux», entre otras cosas para poder usar con facilidad las herramientas vistas en esta sección y preparar el entorno de desarrollo Python. Durante mucho tiempo esto fue difícil de conseguir hasta que *Microsoft* sacó WSL.

WSL[6] nos proporciona una *consola con entorno Linux* que podemos utilizar en nuestro *Windows 10* sin necesidad de instalar una máquina virtual o crear una partición para un

[6] Windows Subsystem for Linux.

Linux nativo. Es importante también saber que existen dos versiones de WSL hoy en día: WSL y WSL2. La segunda es bastante reciente (publicada a mediados de 2019), tiene mejor rendimiento y se adhiere más al comportamiento de un Linux nativo.

Para la instalación de WSL[7] hay que seguir los siguientes pasos:

1. Lanzamos Powershell con permisos de administrador.
2. Activamos la característica de WSL:

```
$ Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-
↪Subsystem-Linux
```

3. Descargamos la imagen de Ubuntu 20.04 que usaremos:

```
$ Invoke-WebRequest -Uri https://aka.ms/wslubuntu2004 -OutFile Ubuntu.appx -
↪UseBasicParsing
```

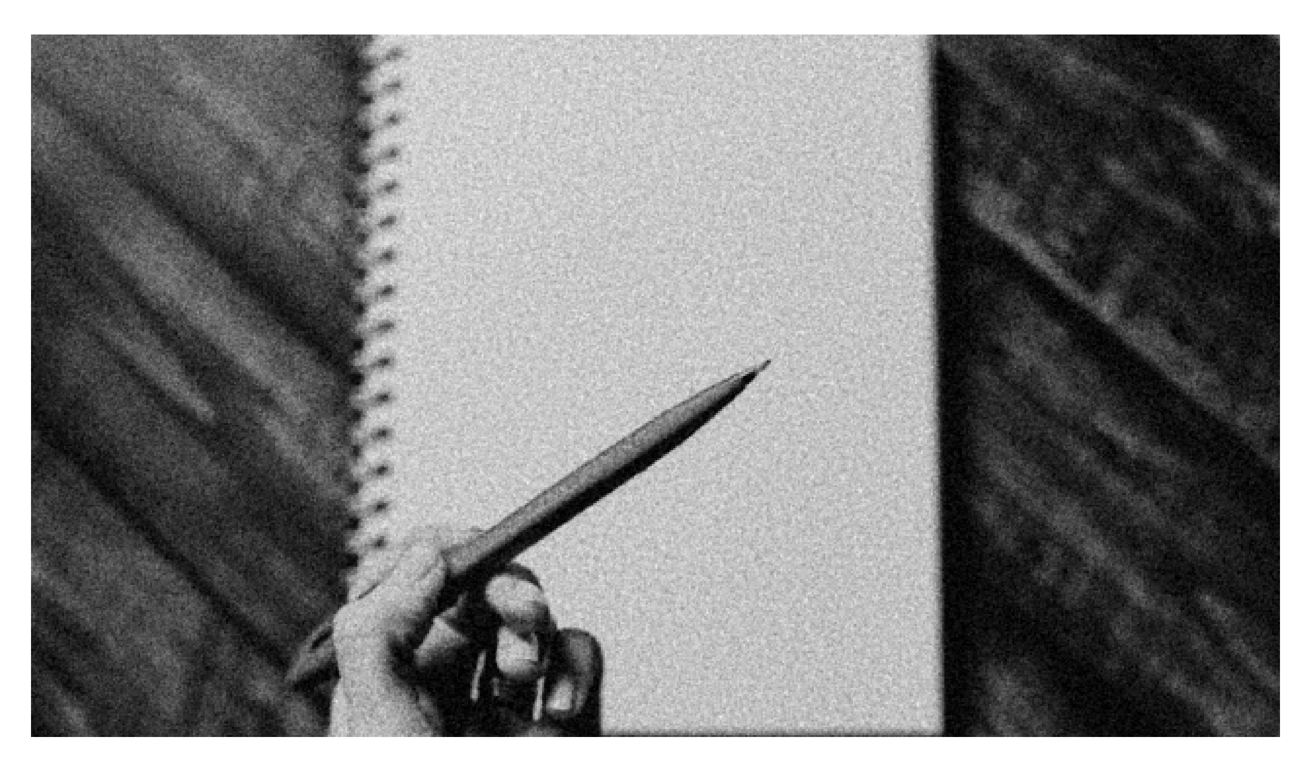
4. Finalmente, la instalamos:

```
$ Add-AppxPackage .\Ubuntu.appx
```

En este punto, WSL debería estar instalado correctamente, y debería también aparecer en el *menú Inicio*.

[7] Tutorial de instalación de WSL.

## 2.3 VSCode



Visual Studio Code (VSCode) es un entorno de desarrollo integrado[1] gratuito y de código abierto que ha ganado mucha relevancia en los últimos años. Permite trabajar fácilmente con multitud de lenguajes de programación y dispone de una gran cantidad de plugins.[2]

### 2.3.1 Atajos de teclado

Conocer los atajos de teclado de tu editor favorito es fundamental para mejorar el flujo de trabajo y ser más productivo. Veamos los principales atajos de teclado de Visual Studio Code[3].

#### Ajustes generales

| Acción | Atajo |
|---|---|
| Abrir paleta de comandos | `Ctrl + Shift + P` |
| Abrir archivo | `Ctrl + P` |
| Nueva ventana | `Ctrl + Shift + N` |

continué en la próxima página

[1] También conocido por IDE siglas en inglés de Integrated Development Environment.
[2] Foto original de portada por Kelly Sikkema en Unsplash.
[3] Fuente: Gastón Danielsen en Dev.To.

Tabla 1 – proviene de la página anterior

| Acción | Atajo |
|---|---|
| Cerrar ventana | `Ctrl + Shift + W` |
| Ajustes del perfil | `Ctrl + ,` |

## Usabilidad

| Acción | Atajo |
|---|---|
| Crear un nuevo archivo | `Ctrl + N` |
| Abrir archivo | `Ctrl + O` |
| Guardar archivo | `Ctrl + S` |
| Cerrar | `Ctrl + F4` |
| Abrir Terminal | `Ctrl + '` |
| Panel de problemas | `Ctrl + Shift + M` |

## Edición básica

| Acción | Atajo |
|---|---|
| Cortar linea | `Ctrl + X` |
| Copiar linea | `Ctrl + C` |
| Borrar linea | `Ctrl + Shift + K` |
| Insertar linea abajo | `Enter` |
| Insertar linea arriba | `Ctrl + Shift + Enter` |
| Buscar en archivo abierto | `Ctrl + F` |
| Reemplazar | `Ctrl + H` |
| Linea de comentario | `Ctrl + /` |
| Bloque de comentario | `Shift + Alt + A` |
| Salto de linea | `Alt + Z` |
| Seleccionar lineas | `Alt + Click Mouse` |
| Tabular linea | `Tab` |
| Destabular linea | `Shift + Tab` |
| Renombrar símbolo | `F2` |

### Pantalla

| Acción | Atajo |
|---|---|
| Acercar Zoom | `Ctrl + +` |
| Alejar Zoom | `Ctrl + -` |
| Barra lateral | `Ctrl + B` |
| Abrir debug | `Ctrl + Shift + D` |
| Panel de salida | `Ctrl + Shift + U` |
| Control de source | `Ctrl + Shift + G` |
| Acceder a extensiones | `Ctrl + Shift + X` |
| Abrir terminal integrado | `Ctrl + Shift + Ñ` |

**Truco:** En macOS 🍏 sustituir `Ctrl` por `Command`.

## 2.3.2 Depurando código

La **depuración de programas** es el proceso de **identificar y corregir errores de programación**. Es conocido también por el término inglés **debugging**, cuyo significado es eliminación de bugs (bichos), manera en que se conoce informalmente a los errores de programación.

Existen varias herramientas de depuración (o *debuggers*). Algunas de ellas en modo texto (terminal) y otras con entorno gráfico (ventanas).

- La herramienta más extendida para **depurar en modo texto** es el módulo pdb (The Python Debugger). Viene incluido en la instalación base de Python y es realmente potente.
- Aunque existen varias herramientas para **depurar en entorno gráfico**, nos vamos a centrar en **Visual Studio Code**.

Lo primero será abrir el fichero (carpeta) donde vamos a trabajar:

### Punto de ruptura

A continuación pondremos un **punto de ruptura** (también llamado **breakpoint**). Esto implica que la ejecución se pare en ese punto y viene indicado por un punto rojo 🔴. Para ponerlo nos tenemos que acercar a la columna que hay a la izquierda del número de línea y hacer clic.

En este ejemplo ponemos un punto de ruptura en la línea 10:

Figura 5: Apertura del fichero a depurar

Figura 6: Punto de ruptura

También es posible añadir **puntos de ruptura condicionales** pulsando con el botón derecho y luego **Add Conditional Breakpoint...**:

Figura 7: Punto de ruptura condicional

### Lanzar la depuración

Ahora ya podemos **lanzar la depuración** pulsando la tecla `F5`. Nos aparecerá el siguiente mensaje en el que dejaremos la opción por defecto **Archivo de Python** y pulsamos la tecla :

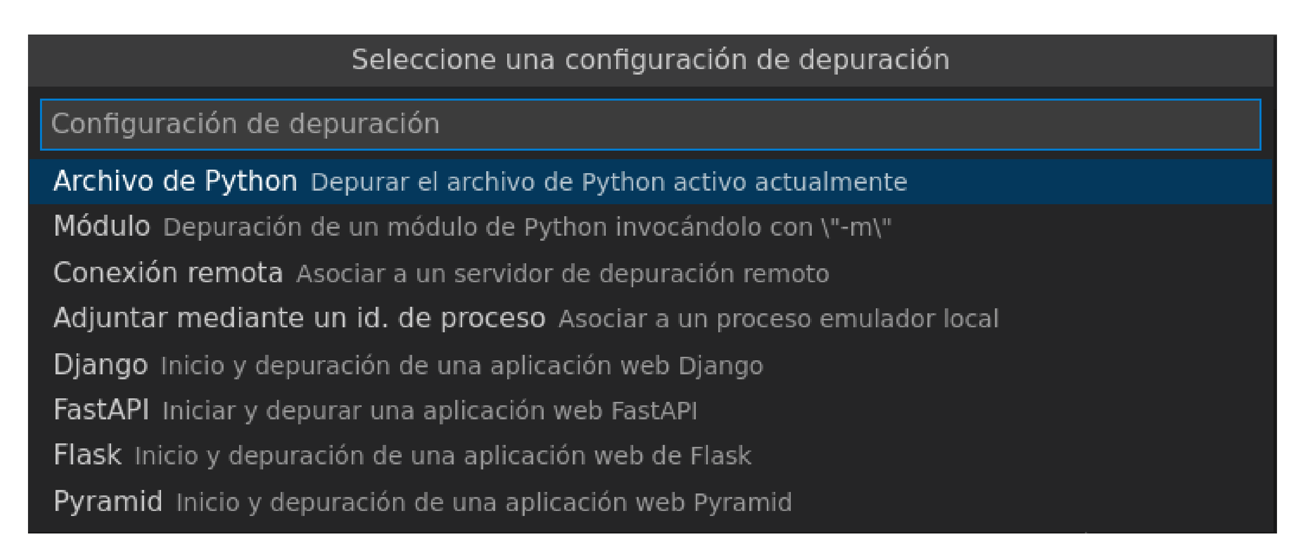



Figura 8: Configuración de la depuración

Ahora ya se inicia el «modo depuración» y veremos una pantalla similar a la siguiente:

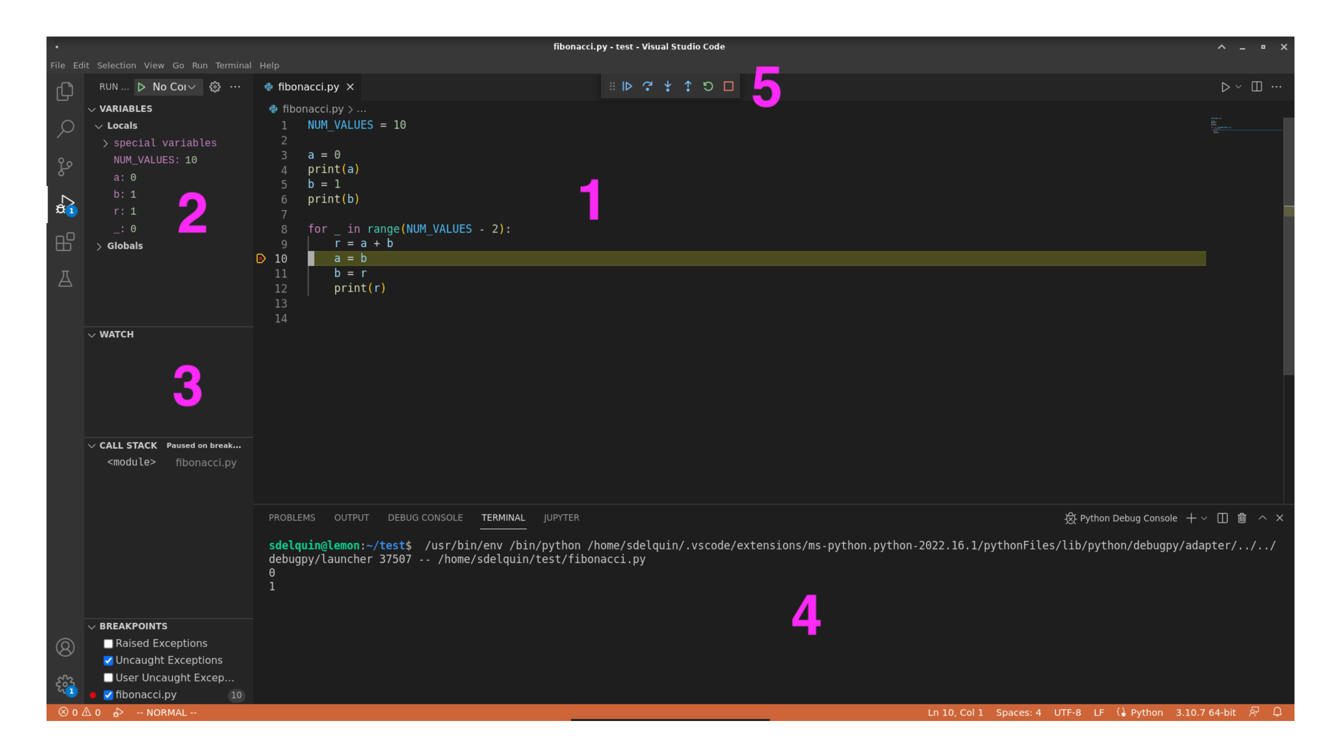



Figura 9: Interfaz en modo depuración

**Zonas de la interfaz en modo depuración:**

1. Código con barra en amarillo que indica la próxima línea que se va a ejecutar.
2. Visualización automática de valores de variables.
3. Visualización personalizada de valores de variables (o expresiones).
4. Salida de la terminal.
5. Barra de herramientas para depuración.

## Controles para la depuración

Veamos con mayor detalle la **barra de herramientas para depuración**:

| **Acción** | Atajo | Significado |
|---|---|---|
| **Continue** | `F5` | Continuar la ejecución del programa hasta el próximo punto de ruptura o hasta su finalización |
| **Step over** | `F10` | Ejecutar la siguiente instrucción del programa |
| **Step into** | `F11` | Ejecutar la siguiente instrucción del programa entrando en un contexto inferior |
| **Step out** | + `F11` | Ejecutar la siguiente instrucción del programa saliendo a un contexto superior |

continué en la próxima página

Tabla 5 – proviene de la página anterior

| Acción | Atajo | Significado |
|---|---|---|
| **Restart** | + + F5 | Reiniciar la depuración del programa |
| **Stop** | + F5 | Detener la depuración del programa |

### Seguimiento de variables

Como hemos indicado previamente, la zona de **Variables** ya nos informa **automáticamente de los valores de las variables** que tengamos en el contexto actual de ejecución:

Pero también es posible **añadir manualmente** el seguimiento de otras variables o expresiones personalizadas desde la zona **Watch**:

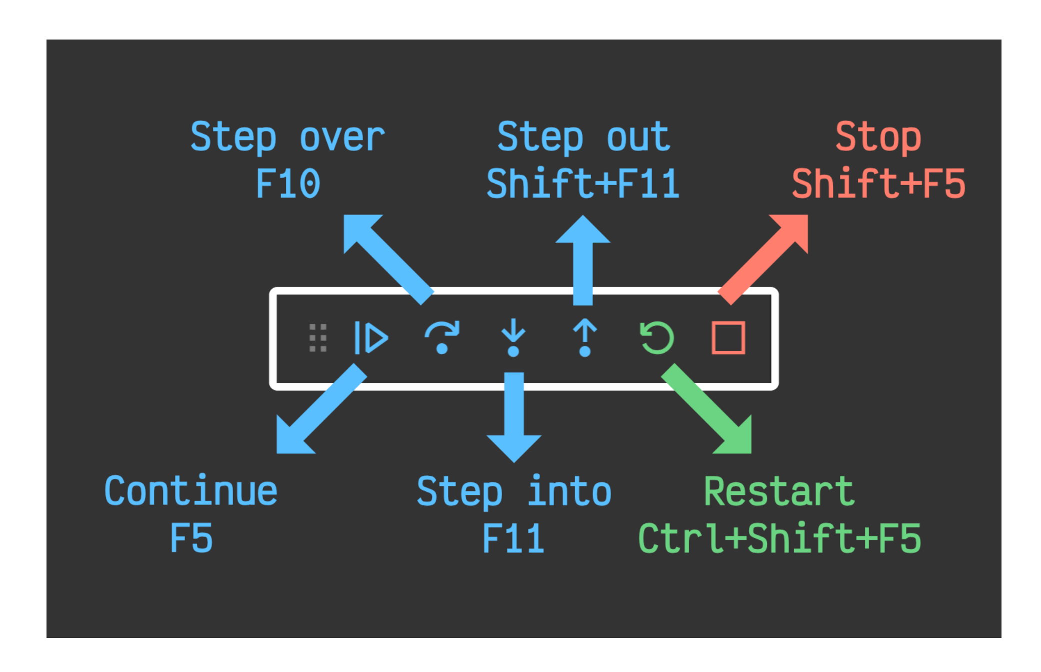



Figura 10: Barra de herarmientas para depuración

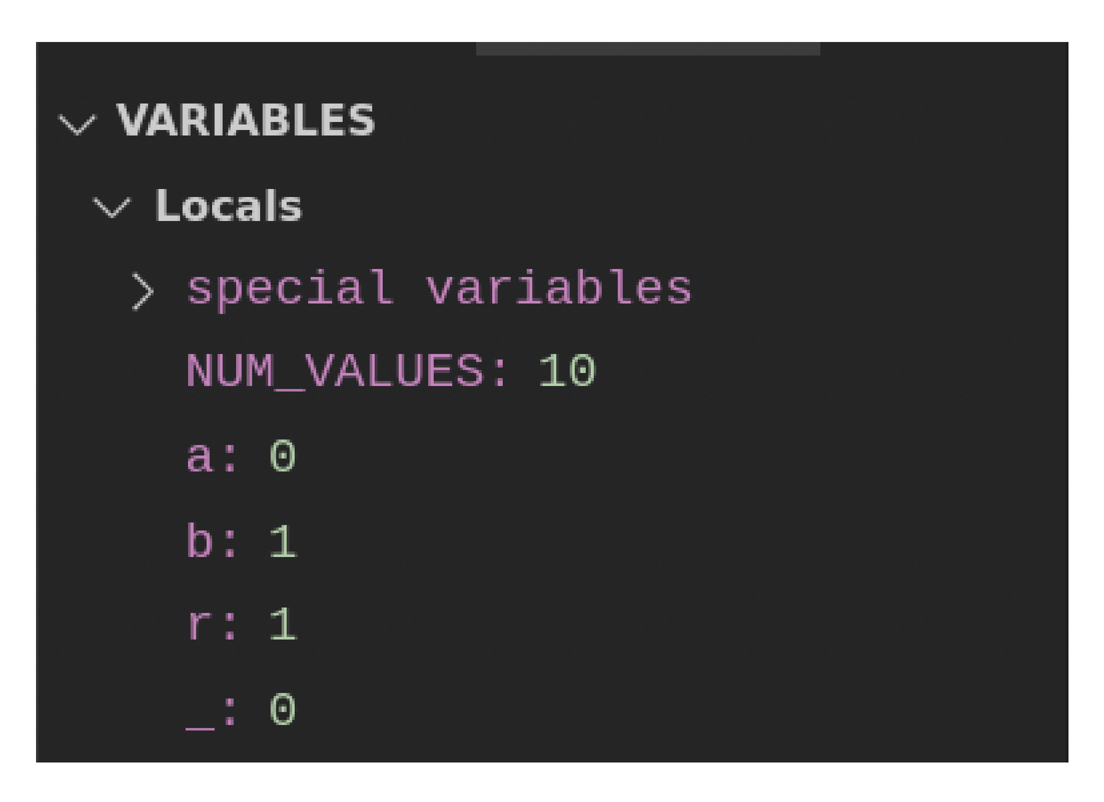



Figura 11: Panel para visualizar variables

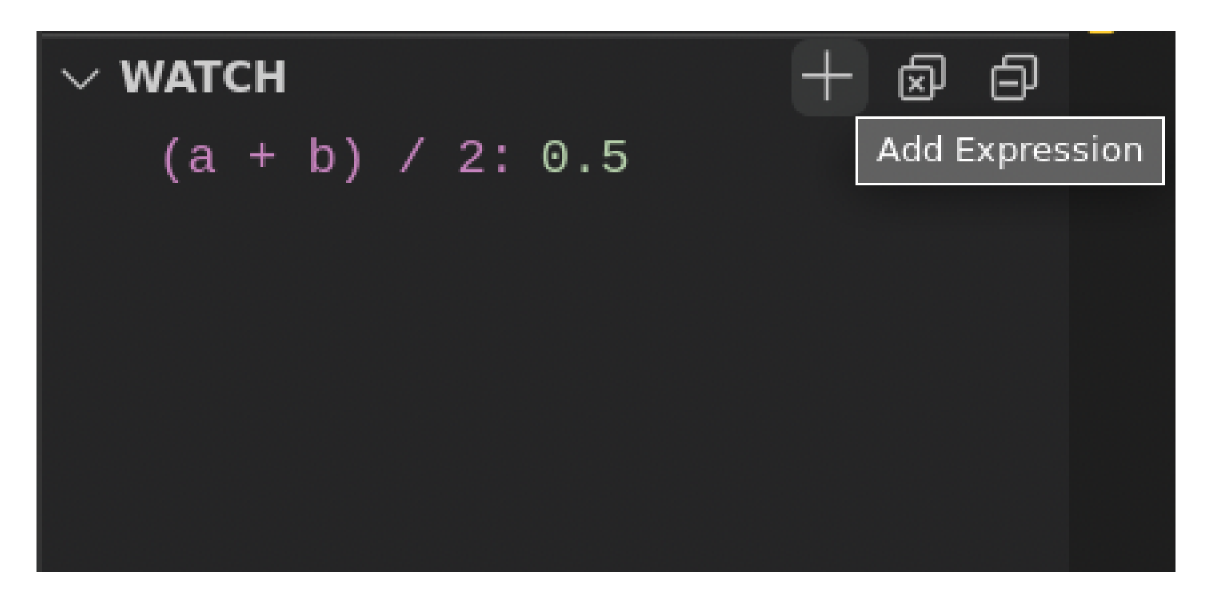



Figura 12: Panel para seguimiento de expresiones

# CAPÍTULO 3

## Tipos de datos

Igual que en el mundo real cada objeto pertenece a una categoría, en programación manejamos objetos que tienen asociado un tipo determinado. En este capítulo se verán los tipos de datos básicos con los que podemos trabajar en Python.

## 3.1 Datos



Los programas están formados por **código** y **datos**. Pero a nivel interno de la memoria del ordenador no son más que una secuencia de bits. La interpretación de estos bits depende del lenguaje de programación, que almacena en la memoria no sólo el puro dato sino distintos metadatos.[1]

Cada «trozo» de memoria contiene realmente un objeto, de ahí que se diga que en Python **todo son objetos**. Y cada objeto tiene, al menos, los siguientes campos:

- Un **tipo** del dato almacenado.
- Un **identificador** único para distinguirlo de otros objetos.
- Un **valor** consistente con su tipo.

### 3.1.1 Tipos de datos

A continuación se muestran los distintos tipos de datos que podemos encontrar en Python, sin incluir aquellos que proveen paquetes externos:

[1] Foto original de portada por Alexander Sinn en Unsplash.

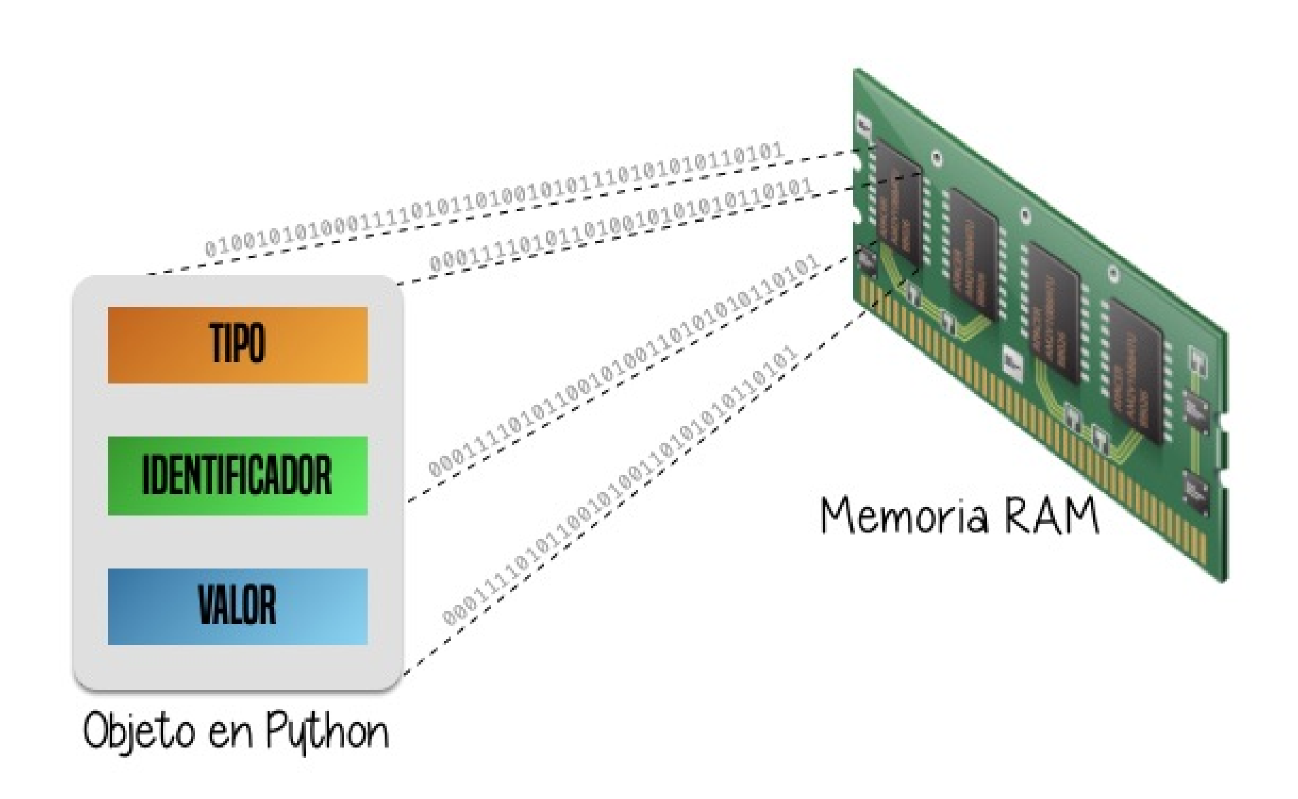



Figura 1: Esquema (*metadatos*) de un objeto en Python

Tabla 1: Tipos de datos en Python

| Nombre | Tipo | Ejemplos |
|---|---|---|
| Booleano | bool | `True`, `False` |
| Entero | int | `21`, `34500`, `34_500` |
| Flotante | float | `3.14`, `1.5e3` |
| Complejo | complex | `2j`, `3 + 5j` |
| Cadena | str | `'tfn'`, `'''tenerife - islas canarias'''` |
| Tupla | tuple | `(1, 3, 5)` |
| Lista | list | `['Chrome', 'Firefox']` |
| Conjunto | set | `set([2, 4, 6])` |
| Diccionario | dict | `{'Chrome': 'v79' , 'Firefox': 'v71'}` |

### 3.1.2 Variables

Las **variables** son fundamentales ya que permiten definir **nombres** para los **valores** que tenemos en memoria y que vamos a usar en nuestro programa.

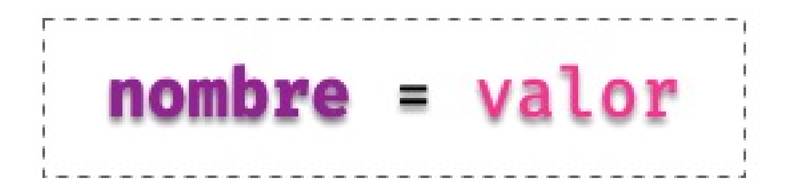



Figura 2: Uso de un *nombre* de variable

#### Reglas para nombrar variables

En Python existen una serie de reglas para los nombres de variables:

1. **Sólo pueden contener los siguientes caracteres**[2]**:**
   - Letras minúsculas.
   - Letras mayúsculas.
   - Dígitos.
   - Guiones bajos (_).
2. Deben **empezar con una letra o un guión bajo**, nunca con un dígito.
3. No pueden ser una **palabra reservada** del lenguaje («keywords»).

[2] Para ser exactos, sí se pueden utilizar otros caracteres, e incluso *emojis* en los nombres de variables, aunque no suele ser una práctica extendida, ya que podría dificultar la legibilidad.

Podemos obtener un listado de las palabras reservadas del lenguaje de la siguiente forma:

```
>>> help('keywords')

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

**Nota:** Por lo general se prefiere dar nombres **en inglés** a las variables que utilicemos, ya que así hacemos nuestro código más «internacional» y con la posibilidad de que otras personas puedan leerlo, entenderlo y – llegado el caso – modificarlo. Es sólo una recomendación, nada impide que se haga en castellano.

**Importante:** Los nombres de variables son «case-sensitive»[3]. Por ejemplo, `stuff` y `Stuff` son nombres diferentes.

### Ejemplos de nombres de variables

Veamos a continuación una tabla con nombres de variables:

Tabla 2: Ejemplos de nombres de variables

| Válido | Inválido | Razón |
|---|---|---|
| `a` | `3` | Empieza por un dígito |
| `a3` | `3a` | Empieza por un dígito |
| `a_b_c___95` | `another-name` | Contiene un carácter no permitido |
| `_abc` | `with` | Es una palabra reservada del lenguaje |
| `_3a` | `3_a` | Empieza por un dígito |

[3] Sensible a cambios en mayúsculas y minúsculas.

## Convenciones para nombres

Mientras se sigan las *reglas* que hemos visto para nombrar variables no hay problema en la forma en la que se escriban, pero sí existe una convención para la **nomenclatura de las variables**. Se utiliza el llamado `snake_case` en el que utilizamos **caracteres en minúsculas** (incluyendo dígitos si procede) junto con **guiones bajos** – cuando sean necesarios para su legibilidad –.[4] Por ejemplo, para nombrar una variable que almacene el número de canciones en nuestro ordenador, podríamos usar `num_songs`.

Esta convención, y muchas otras, están definidas en un documento denominado PEP 8. Se trata de una **guía de estilo** para escribir código en Python. Los PEPs[5] son las propuestas que se hacen para la mejora del lenguaje.

Aunque hay múltiples herramientas disponibles para la comprobación del estilo de código, una bastante accesible es http://pep8online.com/ ya que no necesita instalación, simplemente pegar nuestro código y verificar.

## Constantes

Un caso especial y que vale la pena destacar son las **constantes**. Podríamos decir que es un tipo de variable pero que su valor no cambia a lo largo de nuestro programa. Por ejemplo la velocidad de la luz. Sabemos que su valor es constante de 300.000 km/s. En el caso de las constantes utilizamos **mayúsculas** (incluyendo guiones bajos si es necesario) para nombrarlas. Para la velocidad de la luz nuestra constante se podría llamar: `LIGHT_SPEED`.

## Elegir buenos nombres

Se suele decir que una persona programadora (con cierta experiencia), a lo que dedica más tiempo, es a buscar un buen nombre para sus variables. Quizás pueda resultar algo excesivo pero da una idea de lo importante que es esta tarea. Es fundamental que los nombres de variables sean **autoexplicativos**, pero siempre llegando a un compromiso entre ser concisos y claros.

Supongamos que queremos buscar un nombre de variable para almacenar el número de elementos que se deben manejar en un pedido:

1. `n`
2. `num_elements`
3. `number_of_elements`
4. `number_of_elements_to_be_handled`

[4] Más información sobre convenciones de nombres en PEP 8.
[5] Del término inglés «Python Enhancement Proposals».

No existe una regla mágica que nos diga cuál es el nombre perfecto, pero podemos aplicar el *sentido común* y, a través de la experiencia, ir detectando aquellos nombres que sean más adecuados. En el ejemplo anterior, quizás podríamos descartar de principio la opción *1* y la *4* (por ser demasiado cortas o demasiado largas); nos quedaríamos con las otras dos. Si nos fijamos bien, casi no hay mucha información adicional de la opción *3* con respecto a la *2*. Así que podríamos concluir que la opción *2* es válida para nuestras necesidades. En cualquier caso esto dependerá siempre del contexto del problema que estemos tratando.

Como regla general:

- Usar **nombres** para *variables* (ejemplo `article`).
- Usar **verbos** para *funciones* (ejemplo `get_article()`).
- Usar **adjetivos** para *booleanos* (ejemplo `available`).

### 3.1.3 Asignación

En Python se usa el símbolo `=` para **asignar** un valor a una variable:




Figura 3: Asignación de *valor* a *nombre* de variable

---

**Nota:** Hay que diferenciar la asignación en Python con la igualación en matemáticas. El símbolo `=` lo hemos aprendido desde siempre como una *equivalencia* entre dos *expresiones algebraicas*, sin embargo en Python nos indica una *sentencia de asignación*, del valor (en la derecha) al nombre (en la izquierda).

---

Algunos ejemplos de asignaciones a *variables*:

```
>>> total_population = 157_503
>>> avg_temperature = 16.8
>>> city_name = 'San Cristóbal de La Laguna'
```

Algunos ejemplos de asignaciones a *constantes*:

```
>>> SOUND_SPEED = 343.2
>>> WATER_DENSITY = 997
>>> EARTH_NAME = 'La Tierra'
```

Python nos ofrece la posibilidad de hacer una **asignación múltiple** de la siguiente manera:

```
>>> tres = three = drei = 3
```

En este caso las tres variables utilizadas en el «lado izquierdo» tomarán el valor `3`.

Recordemos que los nombres de variables deben seguir unas *reglas establecidas*, de lo contrario obtendremos un **error sintáctico** del intérprete de Python:

```
>>> 7floor = 40  # el nombre empieza por un dígito
  File "<stdin>", line 1
    7floor = 40
         ^
SyntaxError: invalid syntax

>>> for = 'Bucle'  # el nombre usa la palabra reservada "for"
  File "<stdin>", line 1
    for = 'Bucle'
        ^
SyntaxError: invalid syntax

>>> screen-size = 14  # el nombre usa un carácter no válido
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

### Asignando una variable a otra variable

Las asignaciones que hemos hecho hasta ahora han sido de un **valor literal** a una variable. Pero nada impide que podamos hacer asignaciones de una variable a otra variable:

```
>>> people = 157503
>>> total_population = people
>>> total_population
157503
```

Eso sí, la variable que utilicemos como valor de asignación **debe existir previamente**, ya que si no es así, obtendremos un error informando de que no está definida:

```
>>> total_population = lot_of_people
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lot_of_people' is not defined
```

De hecho, en el *lado derecho* de la asignación pueden aparecer *expresiones* más complejas que se verán en su momento.

### Conocer el valor de una variable

Hemos visto previamente cómo asignar un valor a una variable, pero aún no sabemos cómo «comprobar» el valor que tiene dicha variable. Para ello podemos utilizar dos estrategias:

1. Si estamos en un **intérprete** («shell» o consola) de Python, basta con que usemos el nombre de la variable:

```
>>> final_stock = 38934
>>> final_stock
38934
```

2. Si estamos escribiendo un programa desde el **editor**, debmos hacer uso de `print()`:

```
final_stock = 38934
print(final_stock)
```

**Nota:** `print()` sirve también cuando estamos en una sesión interactiva de Python («shell»)

### Conocer el tipo de una variable

Para poder descubrir el tipo de un literal o una variable, Python nos ofrece la función `type()`. Veamos algunos ejemplos de su uso:

```
>>> type(9)
int

>>> type(1.2)
float

>>> height = 3718
>>> type(height)
int

>>> sound_speed = 343.2
>>> type(sound_speed)
float
```

**Ejercicio**

1. Asigna un valor entero `2001` a la variable `space_odyssey` y muestra su valor.
2. Descubre el tipo del literal `'Good night & Good luck'`.
3. Identifica el tipo del literal `True`.
4. Asigna la expresión `10 * 3.0` a la variable `result` y muestra su tipo.

### 3.1.4 Mutabilidad

**Nivel avanzado**

Las variables son nombres, no lugares. Detrás de esta frase se esconde la reflexión de que cuando asignamos un valor a una variable, lo que realmente está ocurriendo es que se hace **apuntar** el nombre de la variable a una zona de memoria en el que se representa el objeto (con su valor):

```
>>> a = 5
```

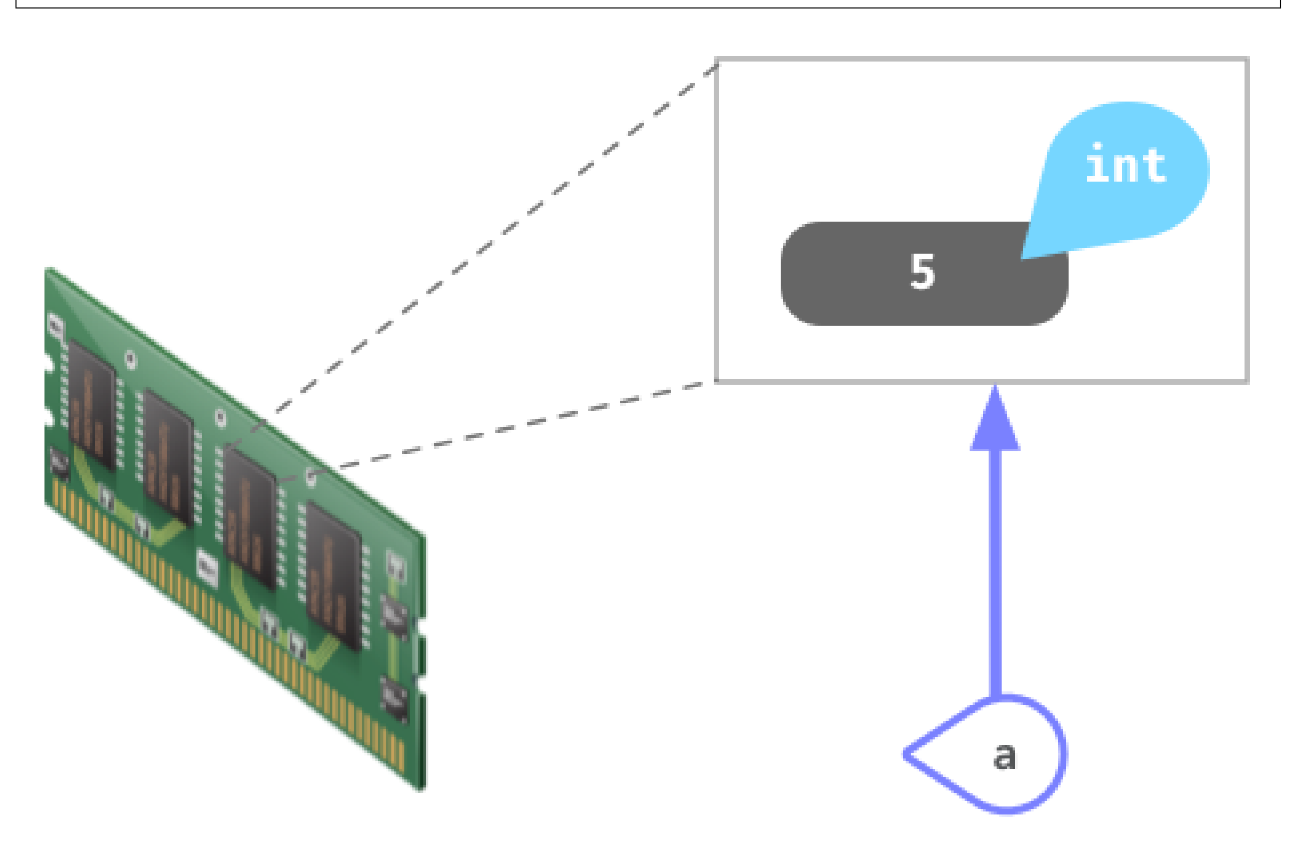



Figura 4: Representación de la asignación de valor a variable

Si ahora «copiamos» el valor de `a` en otra variable `b` se podría esperar que hubiera otro espacio en memoria para dicho valor, pero como ya hemos dicho, son referencias a memoria:

```
>>> b = a
```

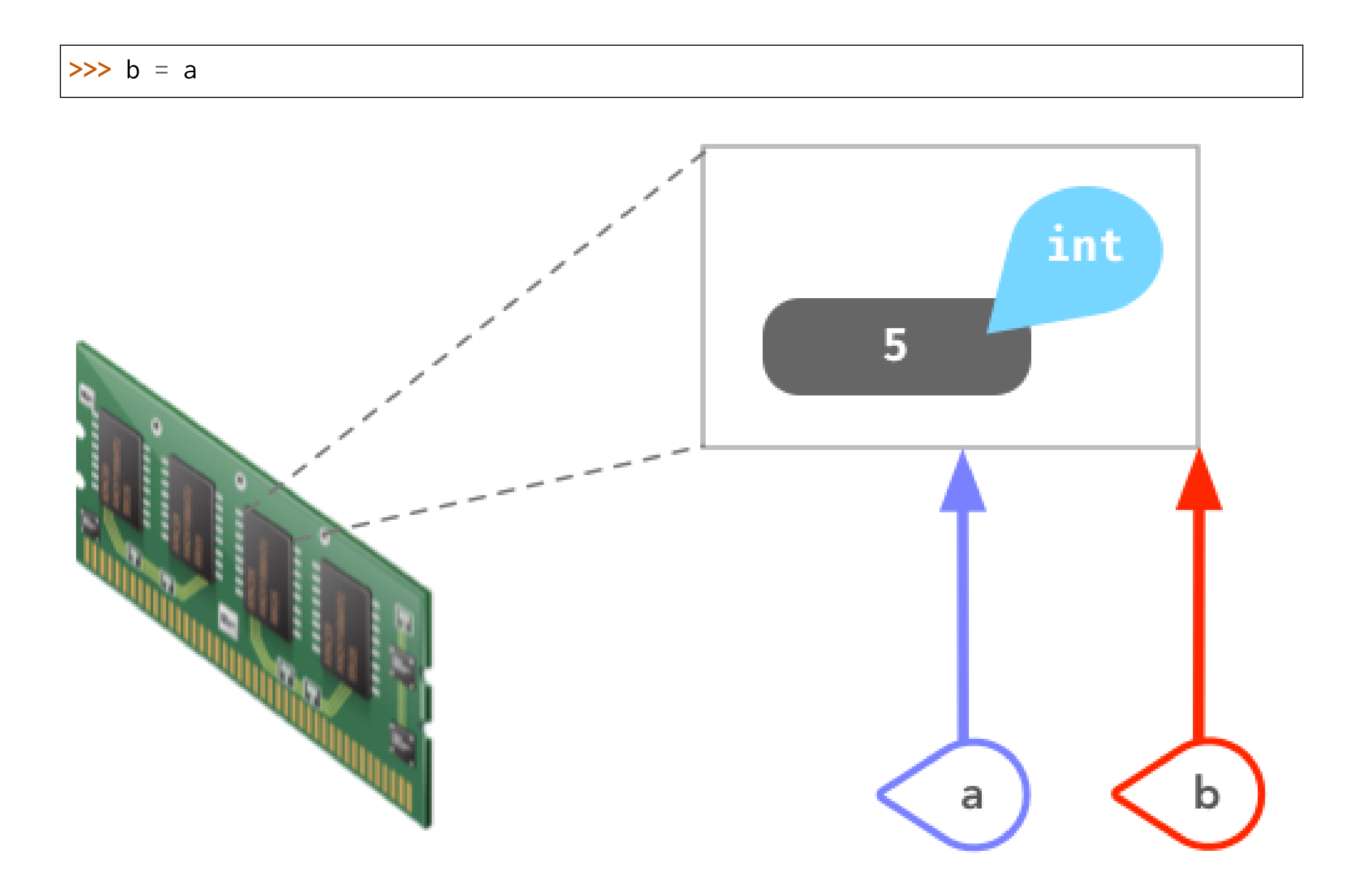



Figura 5: Representación de la asignación de una variable a otra variable

La función `id()` nos permite conocer la dirección de memoria[6] de un objeto en Python. A través de ella podemos comprobar que los dos objetos que hemos creado «apuntan» a la misma zona de memoria:

```
>>> id(a)
4445989712

>>> id(b)
4445989712
```

La prueba de que la zona de memoria no la ocupa el «nombre» de la variable, es que podemos ver cómo se asigna una dirección de memoria únicamente al «valor» literal:

```
>>> id(10)
4333546384

>>> id(20)
4333546704
```

[6] Esto es un detalle de implementación de CPython.

Cada vez que asignamos un nuevo valor a una variable, ésta apunta a una nueva zona de memoria:

```
>>> a = 5
>>> id(a)
4310690224

>>> a = 7
>>> id(a)
4310690288
```

Cuando la zona de memoria que ocupa el objeto se puede modificar hablamos de tipos de datos **mutables**. En otro caso hablamos de tipos de datos **inmutables**.

Por ejemplo, las **listas** son un tipo de dato mutable ya que podemos modificar su contenido (aunque la asignación de un nuevo valor sigue generando un nuevo espacio de memoria).

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/lvCyXeL

Tipos de objetos en Python según su naturaleza de cambio:

| Inmutable | Mutable |
|---|---|
| bool | list |
| int | set |
| float | dict |
| str | |
| tuple | |

**Importante:** El hecho de que un tipo de datos sea inmutable significa que no podemos modificar su valor «in-situ», pero siempre podremos asignarle un nuevo valor (hacerlo apuntar a otra zona de memoria).

### 3.1.5 Funciones «built-in»

#### Nivel intermedio

Hemos ido usando una serie de *funciones* sin ser especialmente conscientes de ello. Esto se debe a que son funciones «built-in» o incorporadas por defecto en el propio lenguaje Python.

Tabla 3: Funciones «built-in»

| | | | | |
|---|---|---|---|---|
| abs() | delattr() | hash() | memoryview() | set() |
| all() | dict() | help() | min() | setattr() |
| any() | any() | hex() | next() | slice() |
| ascii() | divmod() | id() | object() | sorted() |
| bin() | enumerate() | input() | oct() | staticmethod() |
| bool() | eval() | int() | open() | str() |
| breakpoint() | exec() | isinstance() | ord() | sum() |
| bytearray() | filter() | issubclass() | pow() | super() |
| bytes() | float() | iter() | print() | tuple() |
| callable() | format() | len() | property() | type() |
| chr() | frozenset() | list() | range() | vars() |
| classmethod() | getattr() | locals() | repr() | zip() |
| compile() | globals() | map() | reversed() | __import__() |
| complex() | hasattr() | max() | round() | |

Los detalles de estas funciones se puede consultar en la documentación oficial de Python.

### 3.1.6 Pidiendo ayuda

En Python podemos pedir ayuda con la función `help()`.

Supongamos que queremos obtener información sobre `id`. Desde el intérprete de Python ejecutamos lo siguiente:

```
>>> help(id)
Help on built-in function id in module builtins:

id(obj, /)
    Return the identity of an object.

    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object's memory address.)
```

Existe una *forma alternativa* de obtener ayuda: añadiendo el signo de interrogación `?` al término de búsqueda:

```
>>> id?
Signature: id(obj, /)
Docstring:
Return the identity of an object.

This is guaranteed to be unique among simultaneously existing objects.
(CPython uses the object's memory address.)
Type:      builtin_function_or_method
```

### AMPLIAR CONOCIMIENTOS

- Basic Data Types in Python
- Variables in Python
- Immutability in Python

## 3.2 Números



En esta sección veremos los tipos de datos númericos que ofrece Python centrándonos en **booleanos**, **enteros** y **flotantes**.[1]

### 3.2.1 Booleanos

George Boole es considerado como uno de los fundadores del campo de las ciencias de la computación y fue el creador del Álgebra de Boole que da lugar, entre otras estructuras algebraicas, a la Lógica binaria. En esta lógica las variables sólo pueden tomar dos valores discretos: **verdadero** o **falso**.

El tipo de datos `bool` proviene de lo explicado anteriormente y admite dos posibles valores:

[1] Foto original de portada por Brett Jordan en Unsplash.

- `True` que se corresponde con *verdadero* (y también con **1** en su representación numérica).
- `False` que se corresponde con *falso* (y también con **0** en su representación numérica).

Veamos un ejemplo de su uso:

```
>>> is_opened = True
>>> is_opened
True

>>> has_sugar = False
>>> has_sugar
False
```

La primera variable `is_opened` está representando el hecho de que algo esté abierto, y al tomar el valor `True` podemos concluir que sí. La segunda variable `has_sugar` nos indica si una bebida tiene azúcar; dado que toma el valor `False` inferimos que no lleva azúcar.

**Atención:** Tal y como se explicó en *este apartado*, los nombres de variables son «case-sensitive». De igual modo el tipo booleano toma valores `True` y `False` con **la primera letra en mayúsculas**. De no ser así obtendríamos un error sintáctico.

```
>>> is_opened = true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> has_sugar = false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'false' is not defined
```

### 3.2.2 Enteros

Los números enteros no tienen decimales pero sí pueden contener signo y estar expresados en alguna base distinta de la usual (base 10).

#### Literales enteros

Veamos algunos ejemplos de números enteros:

```
>>> 8
8
>>> 0
0
>>> 08
  File "<stdin>", line 1
    08
     ^
SyntaxError: invalid token
>>> 99
99
>>> +99
99
>>> -99
-99
>>> 3000000
3000000
>>> 3_000_000
3000000
```

Dos detalles a tener en cuenta:

- No podemos comenzar un número entero por `0`.
- Python permite dividir los números enteros con *guiones bajos* `_` para clarificar su lectura/escritura. A efectos prácticos es como si esos guiones bajos no existieran.

#### Operaciones con enteros

A continuación se muestra una tabla con las distintas operaciones sobre enteros que podemos realizar en Python:

Tabla 4: Operaciones con enteros en Python

| Operador | Descripción | Ejemplo | Resultado |
|---|---|---|---|
| `+` | Suma | `3 + 9` | `12` |

continué en la próxima página

Tabla 4 – proviene de la página anterior

| Operador | Descripción | Ejemplo | Resultado |
|---|---|---|---|
| `-` | Resta | `6 - 2` | `4` |
| `*` | Multiplicación | `5 * 5` | `25` |
| `/` | División flotante | `9 / 2` | `4.5` |
| `//` | División entera | `9 // 2` | `4` |
| `%` | Módulo | `9 % 4` | `1` |
| `**` | Exponenciación | `2 ** 4` | `16` |

Veamos algunas pruebas de estos operadores:

```
>>> 2 + 8 + 4
14
>>> 4 ** 4
256
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 6 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Es de buen estilo de programación **dejar un espacio** entre cada operador. Además hay que tener en cuenta que podemos obtener errores dependiendo de la operación (más bien de los *operandos*) que estemos utilizando, como es el caso de la *división por cero*.

Igualmente es importante tener en cuenta la **prioridad** de los distintos operadores:

| Prioridad | Operador |
|---|---|
| 1 (mayor) | `()` |
| 2 | `**` |
| 3 | `-a +a` |
| 4 | `* / // %` |
| 5 (menor) | `+ -` |

Ejemplos de prioridad de operadores:

```
>>> 2 ** 2 + 4 / 2
6.0

>>> 2 ** (2 + 4) / 2
32.0
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> 2 ** (2 + 4 / 2)
16.0
```

### Asignación aumentada

Python nos ofrece la posibilidad de escribir una asignación aumentada mezclando la *asignación* y un *operador*.

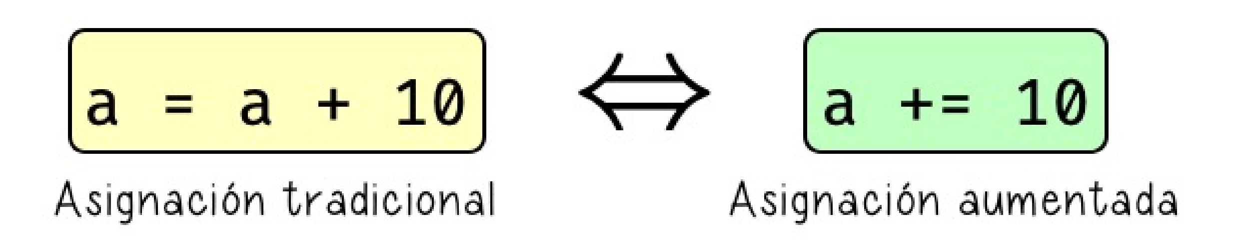



Figura 6: Asignación aumentada en Python

Supongamos que disponemos de 100 vehículos en stock y que durante el pasado mes se han vendido 20 de ellos. Veamos cómo sería el código con asignación tradicional vs. asignación aumentada:

Lista 1: Asignación tradicional

```
>>> total_cars = 100
>>> sold_cars = 20
>>> total_cars = total_cars - sold_cars
>>> total_cars
80
```

Lista 2: Asignación aumentada

```
>>> total_cars = 100
>>> sold_cars = 20
>>> total_cars -= sold_cars
>>> total_cars
80
```

Estas dos formas son equivalentes a nivel de resultados y funcionalidad, pero obviamente tienen diferencias de escritura y legibilidad. De este mismo modo, podemos aplicar un formato compacto al resto de operaciones:

```
>>> random_number = 15

>>> random_number += 5
>>> random_number
20

>>> random_number *= 3
>>> random_number
60

>>> random_number //= 4
>>> random_number
15

>>> random_number **= 1
>>> random_number
15
```

### Módulo

La operación **módulo** (también llamado **resto**), cuyo símbolo en Python es `%`, se define como el resto de dividir dos números. Veamos un ejemplo para enteder bien su funcionamiento:

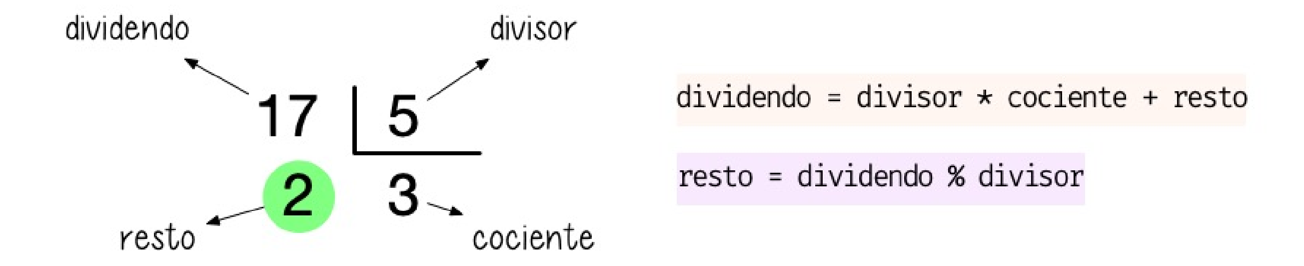



Figura 7: Operador «módulo» en Python

```
>>> dividendo = 17
>>> divisor = 5

>>> cociente = dividendo // divisor  # división entera
>>> resto = dividendo % divisor

>>> cociente
3
>>> resto
2
```

### Exponenciación

Para elevar un número a otro en Python utilizamos el operador de exponenciación `**`:

```
>>> 4 ** 3
64
>>> 4 * 4 * 4
64
```

Se debe tener en cuenta que también podemos elevar un número entero a un **número decimal**. En este caso es como si estuviéramos haciendo una *raíz*[2]. Por ejemplo:

$$4^{\frac{1}{2}} = 4^{0.5} = \sqrt{4} = 2$$

Hecho en Python:

```
>>> 4 ** 0.5
2.0
```

---

**Ejercicio**

pycheck: **quadratic**

---

### Valor absoluto

Python ofrece la función `abs()` para obtener el valor absoluto de un número:

```
>>> abs(-1)
1

>>> abs(1)
1

>>> abs(-3.14)
3.14

>>> abs(3.14)
3.14
```

[2] No siempre es una raíz cuadrada porque se invierten numerador y denominador.

### Límite de un entero

#### Nivel avanzado

¿Cómo de grande puede ser un `int` en Python? La respuesta es **de cualquier tamaño**. Por poner un ejemplo, supongamos que queremos representar un centillón. Este valor viene a ser un «1» seguido por ¡600 ceros! ¿Será capaz Python de almacenarlo?

```
>>> centillion = 10 ** 600

>>> centillion
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

---

**Nota:** En muchos lenguajes tratar con enteros tan largos causaría un «integer overflow». No es el caso de Python que puede manejar estos valores sin problema.

---

¿Qué pasaría si quisiéramos «romper» todas las barreras? Pongamos 10.000 dígitos...

```
>>> 10 ** 10_000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Exceeds the limit (4300) for integer string conversion; use sys.set_int_
↪max_str_digits() to increase the limit
```

Obtenemos un error... pero subsanable, ya que hay forma de ampliar este **límite inicial de 4300 dígitos** usando la función `sys.set_int_max_str_digits()`

## 3.2.3 Flotantes

Los números en **punto flotante**[3] tienen **parte decimal**. Veamos algunos ejemplos de flotantes en Python.

Lista 3: Distintas formas de escribir el flotante *4.0*

```
>>> 4.0
4.0
>>> 4.
4.0
>>> 04.0
4.0
>>> 04.
4.0
```

(continué en la próxima página)

[3] Punto o coma flotante es una notación científica usada por computadores.

(proviene de la página anterior)

```
>>> 4.000_000
4.0
>>> 4e0  # 4.0 * (10 ** 0)
4.0
```

### Conversión de tipos

El hecho de que existan distintos tipos de datos en Python (y en el resto de lenguajes de programación) es una ventaja a la hora de representar la información del mundo real de la mejor manera posible. Pero también se hace necesario buscar mecanismos para convertir unos tipos de datos en otros.

### Conversión implícita

Cuando mezclamos enteros, booleanos y flotantes, Python realiza automáticamente una conversión implícita (o **promoción**) de los valores al tipo de «mayor rango». Veamos algunos ejemplos de esto:

```
>>> True + 25
26
>>> 7 * False
0
>>> True + False
1
>>> 21.8 + True
22.8
>>> 10 + 11.3
21.3
```

Podemos resumir la conversión implícita en la siguiente tabla:

| Tipo 1 | Tipo 2 | Resultado |
|---|---|---|
| `bool` | `int` | `int` |
| `bool` | `float` | `float` |
| `int` | `float` | `float` |

Se puede ver claramente que la conversión numérica de los valores booleanos es:

- `True` 👉 `1`
- `False` 👉 `0`

### Conversión explícita

Aunque más adelante veremos el concepto de **función**, desde ahora podemos decir que existen una serie de funciones para realizar conversiones explícitas de un tipo a otro:

**`bool()`** Convierte el tipo a *booleano*.

**`int()`** Convierte el tipo a *entero*.

**`float()`** Convierte el tipo a *flotante*.

Veamos algunos ejemplos de estas funciones:

```
>>> bool(1)
True
>>> bool(0)
False
>>> int(True)
1
>>> int(False)
0
>>> float(1)
1.0
>>> float(0)
0.0
>>> float(True)
1.0
>>> float(False)
0.0
```

En el caso de que usemos la función `int()` sobre un valor flotante nos retornará su **parte baja**:

$$int(x) = \lfloor x \rfloor$$

Por ejemplo:

```
>>> int(3.1)
3
>>> int(3.5)
3
>>> int(3.9)
3
```

Para **obtener el tipo** de una variable *ya hemos visto* la función `type()`:

```
>>> is_raining = False
>>> type(is_raining)
```

(continué en la próxima página)

(proviene de la página anterior)

```
bool

>>> sound_level = 35
>>> type(sound_level)
int

>>> temperature = 36.6
>>> type(temperature)
float
```

Pero también existe la posibilidad seguimos **comprobar el tipo** que tiene una variable mediante la función `isinstance()`:

```
>>> isinstance(is_raining, bool)
True
>>> isinstance(sound_level, int)
True
>>> isinstance(temperature, float)
True
```

---

**Ejercicio**

pycheck: **sin_approx**

---

## Errores de aproximación

### Nivel intermedio

Supongamos el siguiente cálculo:

```
>>> (19 / 155) * (155 / 19)
    0.9999999999999999
```

Debería dar 1.0, pero no es así puesto que la representación interna de los valores en **coma flotante** sigue el estándar IEEE 754 y estamos trabajando con aritmética finita.

Aunque existen distintas formas de solventar esta limitación, de momento veremos una de las más sencillas utilizando la función «built-in» round() que nos permite redondear un número flotante a un número determinado de decimales:

```
>>> pi = 3.14159265359

>>> round(pi)
```

(continué en la próxima página)

(proviene de la página anterior)

```
3
>>> round(pi, 1)
3.1
>>> round(pi, 2)
3.14
>>> round(pi, 3)
3.142
>>> round(pi, 4)
3.1416
>>> round(pi, 5)
3.14159
```

Para el caso del error de aproximación que nos ocupa:

```
>>> result = (19 / 155) * (155 / 19)

>>> round(result, 1)
1.0
```

**Prudencia:** `round()` aproxima al valor más cercano, mientras que `int()` obtiene siepre el entero «por abajo».

### Límite de un flotante

A diferencia de los *enteros*, los números flotantes sí que tienen un límite en Python. Para descubrirlo podemos ejecutar el siguiente código:

```
>>> import sys

>>> sys.float_info.min
2.2250738585072014e-308

>>> sys.float_info.max
1.7976931348623157e+308
```

### 3.2.4 Bases

**Nivel intermedio**

Los valores numéricos con los que estamos acostumbrados a trabajar están en **base 10** (o decimal). Esto indica que disponemos de 10 «símbolos» para representar las cantidades. En este caso del `0` al `9`.

Pero también es posible representar números en **otras bases**. Python nos ofrece una serie de **prefijos** y **funciones** para este cometido.

#### Base binaria

Cuenta con **2** símbolos para representar los valores: `0` y `1`.

**Prefijo**: `0b`

```
>>> 0b1001
9
>>> 0b1100
12
```

**Función**: `bin()`

```
>>> bin(9)
'0b1001'
>>> bin(12)
'0b1100'
```

#### Base octal

Cuenta con **8** símbolos para representar los valores: `0`, `1`, `2`, `3`, `4`, `5`, `6` y `7`.

**Prefijo**: `0o`

```
>>> 0o6243
3235
>>> 0o1257
687
```

**Función**: `oct()`

```
>>> oct(3235)
'0o6243'
>>> oct(687)
'0o1257'
```

### Base hexadecimal

Cuenta con **16** símbolos para representar los valores: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E` y `F`.

**Prefijo**: `0x`

```
>>> 0x7F2A
32554
>>> 0x48FF
18687
```

**Función**: `hex()`

```
>>> hex(32554)
'0x7f2a'
>>> hex(18687)
'0x48ff'
```

---

**Nota:** Las letras para la representación hexadecimal no atienden a mayúsculas y minúsculas.

---

---

### EJERCICIOS DE REPASO

1. pycheck: **circle_area**
2. pycheck: **sphere_volume**
3. pycheck: **triangle_area**
4. pycheck: **interest_rate**
5. pycheck: **euclid_distance**
6. pycheck: **century_year**
7. pycheck: **red_square**
8. pycheck: **igic**
9. pycheck: **super_fast**
10. pycheck: **move_twice**
11. pycheck: **pillars**
12. pycheck: **clock_time**
13. pycheck: **xor**

## EJERCICIOS EXTERNOS

8. Cat years, dog years
9. Aspect ratio cropping
10. USD => CNY
11. Third angle of a triangle
12. Keep hydrated!
13. Price of mangoes
14. Total pressure calculation
15. NBA full 48 minutes average
16. Age range compatibility equation
17. Formatting decimal places

## AMPLIAR CONOCIMIENTOS

- The Python Square Root Function
- How to Round Numbers in Python
- Operators and Expressions in Python

## 3.3 Cadenas de texto



Las cadenas de texto son **secuencias** de **caracteres**. También se les conoce como «strings» y nos permiten almacenar información textual de forma muy cómoda.[1]

Es importante destacar que Python 3 almacena los caracteres codificados en el estándar Unicode, lo que es una gran ventaja con respecto a versiones antiguas del lenguaje. Además permite representar una cantidad ingente de símbolos incluyendo los famosos emojis 😎.

### 3.3.1 Creando «strings»

Para escribir una cadena de texto en Python basta con rodear los caracteres con comillas simples[6]:

```
>>> 'Mi primera cadena en Python'
'Mi primera cadena en Python'
```

Para incluir *comillas dobles* dentro de la cadena de texto no hay mayor inconveniente:

```
>>> 'Los llamados "strings" son secuencias de caracteres'
'Los llamados "strings" son secuencias de caracteres'
```

[1] Foto original de portada por Roman Kraft en Unsplash.

[6] También es posible utilizar comillas dobles. Yo me he decantado por las comillas simples ya que quedan más limpias y suele ser el formato que devuelve el propio intérprete de Python.

Puede surgir la duda de cómo incluimos *comillas simples* dentro de la propia cadena de texto. Veamos soluciones para ello:

Lista 4: Comillas simples escapadas

```
>>> 'Los llamados \'strings\' son secuencias de caracteres'
"Los llamados 'strings' son secuencias de caracteres"
```

Lista 5: Comillas simples dentro de comillas dobles

```
>>> "Los llamados 'strings' son secuencias de caracteres"
"Los llamados 'strings' son secuencias de caracteres"
```

En la primera opción estamos **escapando** las comillas simples para que no sean tratadas como caracteres especiales. En la segunda opción estamos creando el «string» con comillas dobles (por fuera) para poder incluir directamente las comillas simples (por dentro). Python también nos ofrece esta posibilidad.

### Comillas triples

Hay una forma alternativa de crear cadenas de texto utilizando *comillas triples*. Su uso está pensado principalmente para **cadenas multilínea**:

```
>>> poem = '''To be, or not to be, that is the question:
... Whether 'tis nobler in the mind to suffer
... The slings and arrows of outrageous fortune,
... Or to take arms against a sea of troubles'''
```

**Importante:** Los tres puntos ... que aparecen a la izquierda de las líneas no están incluidos en la cadena de texto. Es el símbolo que ofrece el intérprete de Python cuando saltamos de línea.

### Cadena vacía

La cadena vacía es aquella que no contiene ningún carácter. Aunque a priori no lo pueda parecer, es un recurso importante en cualquier código. Su representación en Python es la siguiente:

```
>>> ''
''
```

### 3.3.2 Conversión

Podemos crear «strings» a partir de otros tipos de datos usando la función `str()`:

```
>>> str(True)
'True'
>>> str(10)
'10'
>>> str(21.7)
'21.7'
```

Para el caso contrario de convertir un «string» a un valor numérico, tenemos a disposición las funciones ya vistas:

```
>>> int('10')
10
>>> float('21.7')
21.7
```

Pero hay que tener en cuenta un detalle. La función `int()` también admite la **base** en la que se encuentra el número. Eso significa que podemos pasar un número, por ejemplo, en **hexadecimal** (como «string») y lo podríamos convertir a su valor entero:

```
>>> int('FF', 16)
255
```

**Nota:** La base por defecto que utiliza `int()` para convertir cadenas de texto es la **base decimal**.

### 3.3.3 Secuencias de escape

Python permite **escapar** el significado de algunos caracteres para conseguir otros resultados. Si escribimos una barra invertida \ antes del carácter en cuestión, le otorgamos un significado especial.

Quizás la *secuencia de escape* más conocida es `\n` que representa un *salto de línea*, pero existen muchas otras:

```
# Salto de línea
>>> msg = 'Primera línea\nSegunda línea\nTercera línea'
>>> print(msg)
Primera línea
Segunda línea
```

(continué en la próxima página)

(proviene de la página anterior)

```
Tercera línea

# Tabulador
>>> msg = 'Valor = \t40'
>>> print(msg)
Valor =     40

# Comilla simple
>>> msg = 'Necesitamos \'escapar\' la comilla simple'
>>> print(msg)
Necesitamos 'escapar' la comilla simple

# Barra invertida
>>> msg = 'Capítulo \\ Sección \\ Encabezado'
>>> print(msg)
Capítulo \ Sección \ Encabezado
```

**Nota:** Al utilizar la función `print()` es cuando vemos realmente el resultado de utilizar los caracteres escapados.

## Expresiones literales

### Nivel intermedio

Hay situaciones en las que nos interesa que los caracteres especiales pierdan ese significado y poder usarlos de otra manera. Existe un modificar de cadena que proporciona Python para tratar el texto *en bruto*. Es el llamado «raw data» y se aplica anteponiendo una `r` a la cadena de texto.

Veamos algunos ejemplos:

```
>>> text = 'abc\ndef'
>>> print(text)
abc
def

>>> text = r'abc\ndef'
>>> print(text)
abc\ndef

>>> text = 'a\tb\tc'
>>> print(text)
a    b    c
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> text = r'a\tb\tc'
>>> print(text)
a\tb\tc
```

**Consejo:** El modificador `r''` es muy utilizado para la escritura de **expresiones regulares**.

### 3.3.4 Más sobre `print()`

Hemos estado utilizando la función `print()` de forma sencilla, pero admite algunos parámetros interesantes:

```
1  >>> msg1 = '¿Sabes por qué estoy acá?'
2  >>> msg2 = 'Porque me apasiona'
3  
4  >>> print(msg1, msg2)
5  ¿Sabes por qué estoy acá? Porque me apasiona
6  
7  >>> print(msg1, msg2, sep='|')
8  ¿Sabes por qué estoy acá?|Porque me apasiona
9  
10 >>> print(msg2, end='!!')
11 Porque me apasiona!!
```

***Línea 4:*** Podemos imprimir todas las variables que queramos separándolas por comas.

***Línea 7:*** El *separador por defecto* entre las variables es un *espacio*, podemos cambiar el carácter que se utiliza como separador entre cadenas.

***Línea 10:*** El *carácter de final de texto* es un *salto de línea*, podemos cambiar el carácter que se utiliza como final de texto.

### 3.3.5 Leer datos desde teclado

Los programas se hacen para tener interacción con el usuario. Una de las formas de interacción es solicitar la entrada de datos por teclado. Como muchos otros lenguajes de programación, Python también nos ofrece la posibilidad de leer la información introducida por teclado. Para ello se utiliza la función `input()`:

```
>>> name = input('Introduzca su nombre: ')
Introduzca su nombre: Sergio
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> name
'Sergio'
>>> type(name)
str

>>> age = input('Introduzca su edad: ')
Introduzca su edad: 41
>>> age
'41'
>>> type(age)
str
```

**Nota:** La función `input()` siempre nos devuelve un objeto de tipo cadena de texto o `str`. Tenerlo muy en cuenta a la hora de trabajar con números, ya que debemos realizar una *conversión explícita*.

**Advertencia:** Aunque está permitido, **NUNCA** llames `input` a una variable porque destruirías la función que nos permite leer datos desde teclado. Y tampoco uses nombres derivados como `_input` o `input_` ya que no son nombres representativos que *identifiquen el propósito de la variable*.

**Ejercicio**

Escriba un programa en Python que lea por teclado dos números enteros y muestre por pantalla el resultado de realizar las operaciones básicas entre ellos.

**Ejemplo**

- Valores de entrada 7 y 4.
- Salida esperada:

```
7+4=11
7-4=3
7*4=28
7/4=1.75
```

**Consejo:** Aproveche todo el potencial que ofrece *print()* para conseguir la salida esperada.

### 3.3.6 Operaciones con «strings»

#### Combinar cadenas

Podemos combinar dos o más cadenas de texto utilizando el operador `+`:

```
>>> proverb1 = 'Cuando el río suena'
>>> proverb2 = 'agua lleva'

>>> proverb1 + proverb2
'Cuando el río suenaagua lleva'

>>> proverb1 + ', ' + proverb2  # incluimos una coma
'Cuando el río suena, agua lleva'
```

#### Repetir cadenas

Podemos repetir dos o más cadenas de texto utilizando el operador `*`:

```
>>> reaction = 'Wow'

>>> reaction * 4
'WowWowWowWow'
```

#### Obtener un carácter

Los «strings» están **indexados** y cada carácter tiene su propia posición. Para obtener un único carácter dentro de una cadena de texto es necesario especificar su **índice** dentro de corchetes `[...]`.

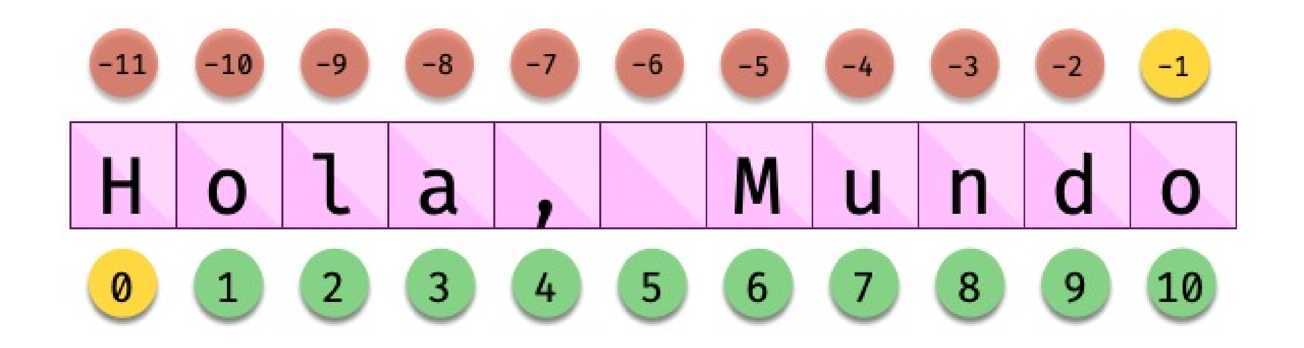



Figura 8: Indexado de una cadena de texto

Veamos algunos ejemplos de acceso a caracteres:

```
>>> sentence = 'Hola, Mundo'

>>> sentence[0]
'H'
>>> sentence[-1]
'o'
>>> sentence[4]
','
>>> sentence[-5]
'M'
```

**Truco:** Nótese que existen tanto **índices positivos** como **índices negativos** para acceder a cada carácter de la cadena de texto. A priori puede parecer redundante, pero es muy útil en determinados casos.

En caso de que intentemos acceder a un índice que no existe, obtendremos un error por *fuera de rango*:

```
>>> sentence[50]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

**Advertencia:** Téngase en cuenta que el indexado de una cadena de texto siempre empieza en **0** y termina en **una unidad menos de la longitud** de la cadena.

Las cadenas de texto son tipos de datos *inmutables*. Es por ello que no podemos modificar un carácter directamente:

```
>>> song = 'Hey Jude'

>>> song[4] = 'D'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

**Truco:** Existen formas de modificar una cadena de texto que veremos más adelante, aunque realmente no estemos transformando el original sino creando un nuevo objeto con las modificaciones.

**Advertencia:** No hay que confundir las *constantes* con los tipos de datos inmutables. Es por ello que las variables que almacenan cadenas de texto, a pesar de ser inmutables, no se escriben en mayúsculas.

## Trocear una cadena

Es posible extraer «trozos» («rebanadas») de una cadena de texto[2]. Tenemos varias aproximaciones para ello:

**`[:]`** Extrae la secuencia entera desde el comienzo hasta el final. Es una especia de **copia** de toda la cadena de texto.

**`[start:]`** Extrae desde `start` hasta el final de la cadena.

**`[:end]`** Extrae desde el comienzo de la cadena hasta `end` *menos 1*.

**`[start:end]`** Extrae desde `start` hasta `end` *menos 1*.

**`[start:end:step]`** Extrae desde `start` hasta `end` *menos 1* haciendo saltos de tamaño `step`.

Veamos la aplicación de cada uno de estos accesos a través de un ejemplo:

```
>>> proverb = 'Agua pasada no mueve molino'

>>> proverb[:]
'Agua pasada no mueve molino'

>>> proverb[12:]
'no mueve molino'

>>> proverb[:11]
'Agua pasada'

>>> proverb[5:11]
'pasada'

>>> proverb[5:11:2]
'psd'
```

**Importante:** El troceado siempre llega a una unidad menos del índice final que hayamos especificado. Sin embargo el comienzo sí coincide con el que hemos puesto.

[2] El término usado en inglés es *slice*.

### Longitud de una cadena

Para obtener la longitud de una cadena podemos hacer uso de `len()`, una función común a prácticamente todos los tipos y estructuras de datos en Python:

```
>>> proverb = 'Lo cortés no quita lo valiente'
>>> len(proverb)
30

>>> empty = ''
>>> len(empty)
0
```

### Pertenencia de un elemento

Si queremos comprobar que una determinada subcadena se encuentra en una cadena de texto utilizamos el operador `in` para ello. Se trata de una expresión que tiene como resultado un valor «booleano» verdadero o falso:

```
>>> proverb = 'Más vale malo conocido que bueno por conocer'

>>> 'malo' in proverb
True

>>> 'bueno' in proverb
True

>>> 'regular' in proverb
False
```

Habría que prestar atención al caso en el que intentamos descubrir si una subcadena **no está** en la cadena de texto:

```
>>> dna_sequence = 'ATGAAATTGAAATGGGA'

>>> not('C' in dna_sequence)  # Primera aproximación
True

>>> 'C' not in dna_sequence  # Forma pitónica
True
```

### Dividir una cadena

Una tarea muy común al trabajar con cadenas de texto es dividirlas por algún tipo de *separador*. En este sentido, Python nos ofrece la función `split()`, que debemos usar anteponiendo el «string» que queramos dividir:

```
>>> proverb = 'No hay mal que por bien no venga'
>>> proverb.split()
['No', 'hay', 'mal', 'que', 'por', 'bien', 'no', 'venga']

>>> tools = 'Martillo,Sierra,Destornillador'
>>> tools.split(',')
['Martillo', 'Sierra', 'Destornillador']
```

**Nota:** Si no se especifica un separador, `split()` usa por defecto cualquier secuencia de espacios en blanco, tabuladores y saltos de línea.

Aunque aún no lo hemos visto, lo que devuelve `split()` es una *lista* (otro tipo de datos en Python) donde cada elemento es una parte de la cadena de texto original:

```
>>> game = 'piedra-papel-tijera'

>>> type(game.split('-'))
list
```

**Ejercicio**

pycheck: **num_words**

Existe una forma algo más avanzada de dividir una cadena a través del **particionado**. Para ello podemos valernos de la función `partition()` que proporciona Python.

Esta función toma un argumento como separador, y divide la cadena de texto en 3 partes: lo que queda a la izquierda del separador, el separador en sí mismo y lo que queda a la derecha del separador:

```
>>> text = '3 + 4'

>>> text.partition('+')
('3 ', '+', ' 4')
```

### Limpiar cadenas

Cuando leemos datos del usuario o de cualquier fuente externa de información, es bastante probable que se incluyan en esas cadenas de texto, *caracteres de relleno*[3] al comienzo y al final. Python nos ofrece la posibilidad de eliminar estos caracteres u otros que no nos interesen.

La función `strip()` se utiliza para eliminar caracteres del principio y del final de un «string». También existen variantes de esta función para aplicarla únicamente al comienzo o únicamente al final de la cadena de texto.

Supongamos que debemos procesar un fichero con números de serie de un determinado artículo. Cada línea contiene el valor que nos interesa pero se han «colado» ciertos caracteres de relleno que debemos limpiar:

```
>>> serial_number = '\n\t   \n 48374983274832    \n\n\t   \t   \n'

>>> serial_number.strip()
'48374983274832'
```

**Nota:** Si no se especifican los caracteres a eliminar, `strip()` usa por defecto cualquier combinación de *espacios en blanco*, *saltos de línea* `\n` y *tabuladores* `\t`.

A continuación vamos a hacer «limpieza» por la izquierda (*comienzo*) y por la derecha (*final*) utilizando la función `lstrip()` y `rstrip()` respectivamente:

Lista 6: «Left strip»

```
>>> serial_number.lstrip()
'48374983274832    \n\n\t   \t   \n'
```

Lista 7: «Right strip»

```
>>> serial_number.rstrip()
'\n\t   \n 48374983274832'
```

Como habíamos comentado, también existe la posibilidad de especificar los caracteres que queremos borrar:

```
>>> serial_number.strip('\n')
'\t   \n 48374983274832    \n\n\t   \t   '
```

**Importante:** La función `strip()` no modifica la cadena que estamos usando (*algo obvio*

[3] Se suele utilizar el término inglés «padding» para referirse a estos caracteres.

*porque los «strings» son inmutables*) sino que devuelve una nueva cadena de texto con las modificaciones pertinentes.

### Realizar búsquedas

Aunque hemos visto que la forma pitónica de saber si una subcadena se encuentra dentro de otra es *a través del operador in*, Python nos ofrece distintas alternativas para realizar búsquedas en cadenas de texto.

Vamos a partir de una variable que contiene un trozo de la canción Mediterráneo de *Joan Manuel Serrat* para ejemplificar las distintas opciones que tenemos:

```
>>> lyrics = '''Quizás porque mi niñez
... Sigue jugando en tu playa
... Y escondido tras las cañas
... Duerme mi primer amor
... Llevo tu luz y tu olor
... Por dondequiera que vaya'''
```

Comprobar si una cadena de texto **empieza o termina por alguna subcadena**:

```
>>> lyrics.startswith('Quizás')
True

>>> lyrics.endswith('Final')
False
```

Encontrar la **primera ocurrencia** de alguna subcadena:

```
>>> lyrics.find('amor')
93

>>> lyrics.index('amor')  # Same behaviour?
93
```

Tanto `find()` como `index()` devuelven el **índice** de la primera ocurrencia de la subcadena que estemos buscando, pero se diferencian en su comportamiento cuando la subcadena buscada no existe:

```
>>> lyrics.find('universo')
-1

>>> lyrics.index('universo')
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Contabilizar el **número de veces que aparece** una subcadena:

```
>>> lyrics.count('mi')
2

>>> lyrics.count('tu')
3

>>> lyrics.count('él')
0
```

**Ejercicio**

pycheck: **lost_word**

## Reemplazar elementos

Podemos usar la función `replace()` indicando la *subcadena a reemplazar*, la *subcadena de reemplazo* y *cuántas instancias* se deben reemplazar. Si no se especifica este último argumento, la sustitución se hará en todas las instancias encontradas:

```
>>> proverb = 'Quien mal anda mal acaba'

>>> proverb.replace('mal', 'bien')
'Quien bien anda bien acaba'

>>> proverb.replace('mal', 'bien', 1)  # sólo 1 reemplazo
'Quien bien anda mal acaba'
```

## Mayúsculas y minúsculas

Python nos permite realizar variaciones en los caracteres de una cadena de texto para pasarlos a mayúsculas y/o minúsculas. Veamos las distintas opciones disponibles:

```
>>> proverb = 'quien a buen árbol se arrima Buena Sombra le cobija'

>>> proverb
'quien a buen árbol se arrima Buena Sombra le cobija'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> proverb.capitalize()
'Quien a buen árbol se arrima buena sombra le cobija'

>>> proverb.title()
'Quien A Buen Árbol Se Arrima Buena Sombra Le Cobija'

>>> proverb.upper()
'QUIEN A BUEN ÁRBOL SE ARRIMA BUENA SOMBRA LE COBIJA'

>>> proverb.lower()
'quien a buen árbol se arrima buena sombra le cobija'

>>> proverb.swapcase()
'QUIEN A BUEN ÁRBOL SE ARRIMA bUENA sOMBRA LE COBIJA'
```

### Identificando caracteres

Hay veces que recibimos información textual de distintas fuentes de las que necesitamos identificar qué tipo de caracteres contienen. Para ello Python nos ofrece un grupo de funciones.

Veamos **algunas** de estas funciones:

Lista 8: Detectar si todos los caracteres son letras o números

```
>>> 'R2D2'.isalnum()
True
>>> 'C3-PO'.isalnum()
False
```

Lista 9: Detectar si todos los caracteres son números

```
>>> '314'.isnumeric()
True
>>> '3.14'.isnumeric()
False
```

Lista 10: Detectar si todos los caracteres son letras

```
>>> 'abc'.isalpha()
True
>>> 'a-b-c'.isalpha()
```

(continué en la próxima página)

(proviene de la página anterior)

```
False
```

Lista 11: Detectar mayúsculas/minúsculas

```
>>> 'BIG'.isupper()
True
>>> 'small'.islower()
True
>>> 'First Heading'.istitle()
True
```

### 3.3.7 Interpolación de cadenas

En este apartado veremos cómo **interpolar** valores dentro de cadenas de texto utilizando diferentes formatos. Interpolar (en este contexto) significa sustituir una variable por su valor dentro de una cadena de texto.

Veamos los estilos que proporciona Python para este cometido:

| Nombre | Símbolo | Soportado |
|---|---|---|
| Estilo antiguo | `%` | >= Python2 |
| Estilo «nuevo» | `.format` | >= Python2.6 |
| «f-strings» | `f''` | >= Python3.6 |

Aunque aún podemos encontrar código con el estilo antiguo y el estilo nuevo en el formateo de cadenas, vamos a centrarnos en el análisis de los **«f-strings»** que se están utilizando bastante en la actualidad.

#### «f-strings»

Los **f-strings** aparecieron en Python 3.6 y se suelen usar en código de nueva creación. Es la forma más potente – y en muchas ocasiones más eficiente – de formar cadenas de texto incluyendo valores de otras variables.

La **interpolación** en cadenas de texto es un concepto que existe en la gran mayoría de lenguajes de programación y hace referencia al hecho de sustituir los nombres de variables por sus valores cuando se construye un «string».

Para indicar en Python que una cadena es un «f-string» basta con precederla de una `f` e incluir las variables o expresiones a interpolar entre llaves `{...}`.

Supongamos que disponemos de los datos de una persona y queremos formar una frase de bienvenida con ellos:

```
>>> name = 'Elon Musk'
>>> age = 49
>>> fortune = 43_300

>>> f'Me llamo {name}, tengo {age} años y una fortuna de {fortune} millones'
'Me llamo Elon Musk, tengo 49 años y una fortuna de 43300 millones'
```

> **Advertencia:** Si olvidamos poner la **f** delante del «string» no conseguiremos sustitución de variables.

Podría surgir la duda de cómo incluir llaves dentro de la cadena de texto, teniendo en cuenta que las llaves son símbolos especiales para la interpolación de variables. La respuesta es duplicar las llaves:

```
>>> x = 10

>>> f'The variable is {{ x = {x} }}'
'The variable is { x = 10 }'
```

## Formateando cadenas

### Nivel intermedio

Los «f-strings» proporcionan una gran variedad de **opciones de formateado**: ancho del texto, número de decimales, tamaño de la cifra, alineación, etc. Muchas de estas facilidades se pueden consultar en el artículo Best of Python3.6 f-strings[4]

**Dando formato a valores enteros:**

```
>>> mount_height = 3718

>>> f'{mount_height:10d}'
'      3718'

>>> f'{mount_height:010d}'
'0000003718'
```

**Dando formato a otras bases:**

```
>>> value = 0b10010011
```

(continué en la próxima página)

[4] Escrito por Nirant Kasliwal en Medium.

(proviene de la página anterior)

```
>>> f'{value}'
'147'
>>> f'{value:b}'
'10010011'

>>> value = 0o47622
>>> f'{value}'
'20370'
>>> f'{value:o}'
'47622'

>>> value = 0xab217
>>> f'{value}'
'700951'
>>> f'{value:x}'
'ab217'
```

**Dando formato a valores flotantes:**

```
>>> pi = 3.14159265

>>> f'{pi:f}'  # 6 decimales por defecto (se rellenan con ceros si procede)
'3.141593'

>>> f'{pi:.3f}'
'3.142'

>>> f'{pi:12f}'
'    3.141593'

>>> f'{pi:7.2f}'
'   3.14'

>>> f'{pi:07.2f}'
'0003.14'

>>> f'{pi:.010f}'
'3.1415926500'

>>> f'{pi:e}'
'3.141593e+00'
```

**Alineando valores:**

```
>>> text1 = 'how'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> text2 = 'are'
>>> text3 = 'you'

>>> f'{text1:<7s}|{text2:^11s}|{text3:>7s}'
'how    |    are    |    you'

>>> f'{text1:-<7s}|{text2:·^11s}|{text3:->7s}'
'how----|····are····|----you'
```

### Modo «debug»

A partir de Python 3.8, los «f-strings» permiten imprimir el nombre de la variable y su valor, como un atajo para depurar nuestro código. Para ello sólo tenemos que incluir un símbolo `=` después del nombre de la variable:

```
>>> serie = 'The Simpsons'
>>> imdb_rating = 8.7
>>> num_seasons = 30

>>> f'{serie=}'
"serie='The Simpsons'"

>>> f'{imdb_rating=}'
'imdb_rating=8.7'

>>> f'{serie[4:]=}'  # incluso podemos añadir expresiones!
"serie[4:]='Simpsons'"

>>> f'{imdb_rating / num_seasons=}'
'imdb_rating / num_seasons=0.29'
```

### Modo «representación»

Si imprimimos el valor de una variable utilizando un «f-string», obviamente veremos ese valor tal y como esperaríamos:

```
>>> name = 'Steven Spielberg'

>>> print(f'{name}')
Steven Spielberg
```

Pero si quisiéramos ver la **representación** del objeto, tal y como se almacena internamente, podríamos utilizar el modificador `!r` en el «f-string»:

```
>>> name = 'Steven Spielberg'

>>> print(f'{name!r}')
'Steven Spielberg'
```

En este caso se han añadido las comillas denotando que es una cadena de texto. Este modificador se puede aplicar a cualquier otro tipo de dato.

---

**Ejercicio**

Dada la variable:

```
e = 2.71828
```

, obtenga los siguientes resultados utilizando «f-strings»:

```
'2.718'
'2.718280'
'    2.72'  # 4 espacios en blanco
'2.718280e+00'
'00002.7183'
'            2.71828'  # 12 espacios en blanco
```

---

### 3.3.8 Caracteres Unicode

Python trabaja *por defecto* con caracteres **Unicode**. Eso significa que tenemos acceso a la amplia carta de caracteres que nos ofrece este estándar de codificación.

Supongamos un ejemplo sobre el típico «emoji» de un **cohete** definido en este cuadro:

Figura 9: Representación Unicode del carácter ROCKET

La función `chr()` permite representar un carácter **a partir de su código**:

```
>>> rocket_code = 0x1F680
>>> rocket = chr(rocket_code)
>>> rocket
'🚀'
```

La función `ord()` permite obtener el código (decimal) de un carácter **a partir de su representación**:

```
>>> rocket_code = hex(ord(rocket))
>>> rocket_code
'0x1f680'
```

El modificador `\N` permite representar un carácter **a partir de su nombre**:

```
>>> '\N{ROCKET}'
'🚀'
```

**Ver también:**

Tabla ASCII

### Comparar cadenas

Cuando comparamos dos cadenas de texto lo hacemos en términos **lexicográficos**. Es decir, se van comparando los caracteres de ambas cadenas uno a uno y se va mirando cuál está «antes».

Por ejemplo:

```
>>> 'arca' < 'arpa' # 'ar' es igual para ambas
True

>>> ord('c')
99
>>> ord('p')
112
```

**Nota:** Internamente se utiliza la función `ord()` para comparar qué carácter está «antes».

Otros ejemplos:

```
>>> 'a' < 'antes'
True
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 'antes' < 'después'
True

>>> 'después' < 'ahora'
False

>>> 'ahora' < 'a'
False
```

Tener en cuenta que en Python la letras mayúsculas van antes que las minúsculas:

```
>>> 'A' < 'a'
True

>>> ord('A')
65
>>> ord('a')
97
```

### 3.3.9 Casos de uso

**Nivel avanzado**

Hemos estado usando muchas funciones de objetos tipo «string» (y de otros tipos previamente). Pero quizás no sabemos aún como podemos descubrir todo lo que podemos hacer con ellos y los **casos de uso** que nos ofrece.

Python proporciona una *función «built-in»* llamada `dir()` para inspeccionar un determinado tipo de objeto:

```
>>> text = 'This is it!'

>>> dir(text)
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'__getnewargs__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
```

(continué en la próxima página)

(proviene de la página anterior)

```
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Esto es aplicable tanto a variables como a literales e incluso a tipos de datos (clases) explícitos:

```
>>> dir(10)
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 ...
 'imag',
 'numerator',
 'real',
 'to_bytes']

>>> dir(float)
['__abs__',
 '__add__',
 '__bool__',
 '__class__',
 ...
 'hex',
 'imag',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'is_integer',
'real']
```

## EJERCICIOS DE REPASO

1. pycheck: **switch_name**
2. pycheck: **samba_split**
3. pycheck: **nif_digit**
4. pycheck: **n_repeat**
5. pycheck: **str_metric**
6. pycheck: **h2md**
7. pycheck: **count_sheeps**
8. pycheck: **strip1**
9. pycheck: **swap_name**
10. pycheck: **find_integral**
11. pycheck: **multiply_jack**

## AMPLIAR CONOCIMIENTOS

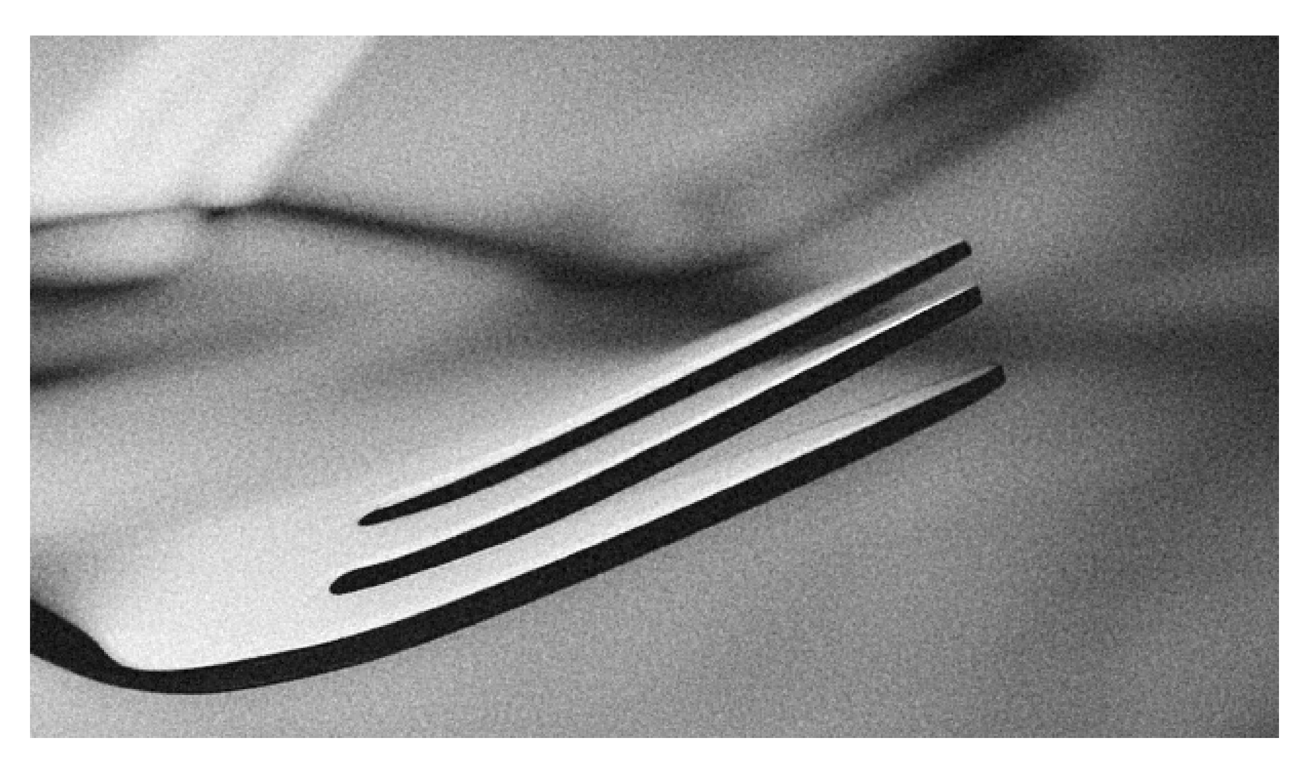
- A Guide to the Newer Python String Format Techniques
- Strings and Character Data in Python
- How to Convert a Python String to int
- Your Guide to the Python print<> Function
- Basic Input, Output, and String Formatting in Python
- Unicode & Character Encodings in Python: A Painless Guide
- Python String Formatting Tips & Best Practices
- Python 3's f-Strings: An Improved String Formatting Syntax
- Splitting, Concatenating, and Joining Strings in Python
- Conditional Statements in Python
- Python String Formatting Best Practices

# CAPÍTULO 4

## Control de flujo

Todo programa informático está formado por *instrucciones* que se ejecutan en forma secuencial de «arriba» a «abajo», de igual manera que leeríamos un libro. Este orden constituye el llamado **flujo** del programa. Es posible modificar este flujo secuencial para que tome *bifurcaciones* o *repita* ciertas instrucciones. Las sentencias que nos permiten hacer estas modificaciones se engloban en el *control de flujo*.

## 4.1 Condicionales



En esta sección veremos las sentencias `if` y `match-case` junto a las distintas variantes que pueden asumir, pero antes de eso introduciremos algunas cuestiones generales de *escritura de código*.[1]

### 4.1.1 Definición de bloques

A diferencia de otros lenguajes que utilizan llaves para definir los bloques de código, cuando Guido Van Rossum *creó el lenguaje* quiso evitar estos caracteres por considerarlos innecesarios. Es por ello que en Python los bloques de código se definen a través de **espacios en blanco, preferiblemente 4**.[2] En términos técnicos se habla del **tamaño de indentación**.

---

**Consejo:** Esto puede resultar extraño e incómodo a personas que vienen de otros lenguajes de programación pero desaparece rápido y se siente natural a medida que se escribe código.

---

[1] Foto original de portada por ali nafezarefi en Unsplash.

[2] Reglas de indentación definidas en PEP 8

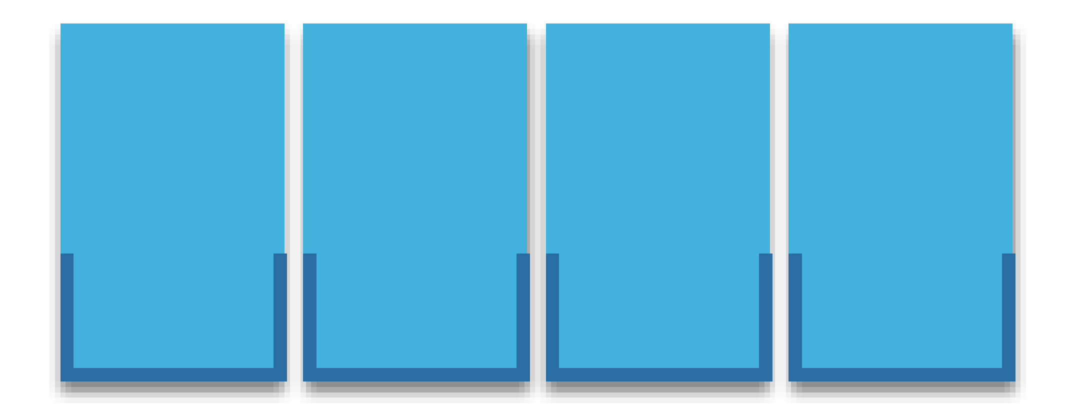


Figura 1: Python recomienda 4 espacios en blanco para indentar

### 4.1.2 Comentarios

Los comentarios son anotaciones que podemos incluir en nuestro programa y que nos permiten aclarar ciertos aspectos del código. Estas indicaciones son ignoradas por el intérprete de Python.

Los comentarios se incluyen usando el símbolo almohadilla # y comprenden hasta el final de la línea.

Lista 1: Comentario en bloque

```
# Universe age expressed in days
universe_age = 13800 * (10 ** 6) * 365
```

Los comentarios también pueden aparecer en la misma línea de código, aunque la guía de estilo de Python no aconseja usarlos en demasía:

Lista 2: Comentario en línea

```
stock = 0   # Release additional articles
```

Reglas para escribir buenos comentarios:[6]

1. Los comentarios no deberían duplicar el código.
2. Los buenos comentarios no arreglan un código poco claro.
3. Si no puedes escribir un comentario claro, puede haber un problema en el código.
4. Los comentarios deberían evitar la confusión, no crearla.
5. Usa comentarios para explicar código no idiomático.
6. Proporciona enlaces a la fuente original del código copiado.

[6] Referencia: Best practices for writing code comments

7. Incluye enlaces a referencias externas que sean de ayuda.
8. Añade comentarios cuando arregles errores.
9. Usa comentarios para destacar implementaciones incompletas.

### 4.1.3 Ancho del código

Los programas suelen ser más legibles cuando las líneas no son excesivamente largas. La longitud máxima de línea recomendada por la guía de estilo de Python es de **80 caracteres**.

Sin embargo, esto genera una cierta polémica hoy en día, ya que los tamaños de pantalla han aumentado y las resoluciones son mucho mayores que hace años. Así las líneas de más de 80 caracteres se siguen visualizando correctamente. Hay personas que son más estrictas en este límite y otras más flexibles.

En caso de que queramos **romper una línea de código** demasiado larga, tenemos dos opciones:

1. Usar la *barra invertida* `\`:

```
>>> factorial = 4 * 3 * 2 * 1

>>> factorial = 4 * \
...             3 * \
...             2 * \
...             1
```

2. Usar los *paréntesis* `(...)`:

```
>>> factorial = 4 * 3 * 2 * 1

>>> factorial = (4 *
...              3 *
...              2 *
...              1)
```

### 4.1.4 La sentencia `if`

La sentencia condicional en Python (al igual que en muchos otros lenguajes de programación) es `if`. En su escritura debemos añadir una **expresión de comparación** terminando con dos puntos al final de la línea. Veamos un ejemplo:

```
>>> temperature = 40
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> if temperature > 35:
...     print('Aviso por alta temperatura')
...
Aviso por alta temperatura
```

**Nota:** Nótese que en Python no es necesario incluir paréntesis `(` y `)` al escribir condiciones. Hay veces que es recomendable por claridad o por establecer prioridades.

En el caso anterior se puede ver claramente que la condición se cumple y por tanto se ejecuta la instrucción que tenemos dentro del cuerpo de la condición. Pero podría no ser así. Para controlar ese caso existe la sentencia `else`. Veamos el mismo ejemplo anterior pero añadiendo esta variante:

```
>>> temperature = 20

>>> if temperature > 35:
...     print('Aviso por alta temperatura')
... else:
...     print('Parámetros normales')
...
Parámetros normales
```

Podríamos tener incluso condiciones dentro de condiciones, lo que se viene a llamar técnicamente **condiciones anidadas**[3]. Veamos un ejemplo ampliando el caso anterior:

```
>>> temperature = 28

>>> if temperature < 20:
...     if temperature < 10:
...         print('Nivel azul')
...     else:
...         print('Nivel verde')
... else:
...     if temperature < 30:
...         print('Nivel naranja')
...     else:
...         print('Nivel rojo')
...
Nivel naranja
```

Python nos ofrece una mejora en la escritura de condiciones anidadas cuando aparecen consecutivamente un `else` y un `if`. Podemos sustituirlos por la sentencia `elif`:

[3] El anidamiento (o «nesting») hace referencia a incorporar sentencias unas dentro de otras mediante la inclusión de diversos niveles de profunidad (indentación).




Figura 2: Construcción de la sentencia `elif`

Apliquemos esta mejora al código del ejemplo anterior:

```
>>> temperature = 28

>>> if temperature < 20:
...     if temperature < 10:
...         print('Nivel azul')
...     else:
...         print('Nivel verde')
... elif temperature < 30:
...     print('Nivel naranja')
... else:
...     print('Nivel rojo')
...
Nivel naranja
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/wd58B4t

### 4.1.5 Asignaciones condicionales

#### Nivel intermedio

Supongamos que queremos asignar un nivel de riesgo de incendio en función de la temperatura. En su **versión clásica** escribiríamos:

```
>>> temperature = 35

>>> if temperature < 30:
...     fire_risk = 'LOW'
... else:
...     fire_risk = 'HIGH'
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> fire_risk
'HIGH'
```

Sin embargo, esto lo podríamos abreviar con una **asignación condicional de una única línea**:

```
>>> fire_risk = 'LOW' if temperature < 30 else 'HIGH'

>>> fire_risk
'HIGH'
```

### 4.1.6 Operadores de comparación

Cuando escribimos condiciones debemos incluir alguna expresión de comparación. Para usar estas expresiones es fundamental conocer los operadores que nos ofrece Python:

| Operador | Símbolo |
|---|---|
| Igualdad | == |
| Desigualdad | != |
| Menor que | < |
| Menor o igual que | <= |
| Mayor que | > |
| Mayor o igual que | >= |

A continuación vamos a ver una serie de ejemplos con expresiones de comparación. Téngase en cuenta que estas expresiones habría que incluirlas dentro de la sentencia condicional en el caso de que quisiéramos tomar una acción concreta:

```
# Asignación de valor inicial
>>> value = 8

>>> value == 8
True

>>> value != 8
False

>>> value < 12
True

>>> value <= 7
False
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> value > 4
True

>>> value >= 9
False
```

Python ofrece la posibilidad de ver si un valor está entre dos límites de manera directa. Así, por ejemplo, para descubrir si `x` está entre *4* y *12* haríamos:

```
>>> 4 <= x <= 12
True
```

### 4.1.7 Operadores lógicos

**Podemos escribir condiciones más complejas usando los operadores lógicos:**

- `and`
- `or`
- `not`

```
# Asignación de valor inicial
>>> x = 8

>>> x > 4 or x > 12  # True or False
True

>>> x < 4 or x > 12  # False or False
False

>>> x > 4 and x > 12  # True and False
False

>>> x > 4 and x < 12  # True and True
True

>>> not(x != 8)  # not False
True
```

Véanse las **tablas de la verdad** para cada operador lógico:

**Nota:**

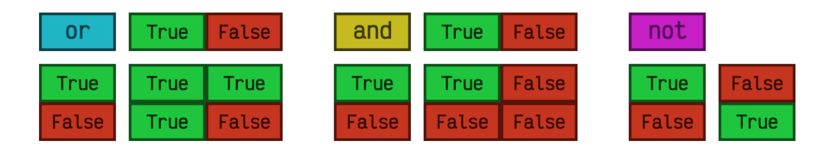



Figura 3: Resultados al aplicar operadores lógicos

1. Una expresión de comparación siempre devuelve un valor *booleano*, es decir `True` o `False`.
2. El uso de paréntesis, en función del caso, puede aclarar la expresión de comparación.

---

**Ejercicio**

Dada una variable `year` con un valor entero, compruebe si dicho año es **bisiesto** o no lo es.

ℹ️ Un año es bisiesto en el calendario Gregoriano, si es divisible entre 4 y no divisible entre 100, o bien si es divisible entre 400. Puedes hacer la comprobación en esta lista de años bisiestos.

**Ejemplo**

- Entrada: `2008`
- Salida: `Es un año bisiesto`

---

## Cortocircuito lógico

Es interesante comprender que **las expresiones lógicas no se evalúan por completo si se dan una serie de circunstancias**. Aquí es donde entra el concepto de **cortocircuito** que no es más que una forma de denominar a este escenario.

Supongamos un ejemplo en el que utilizamos un **teléfono móvil** que mide la batería por la variable `power` de 0 a 100% y la cobertura 4G por la variable `signal_4g` de 0 a 100%.

Para poder **enviar un mensaje por Telegram** necesitamos tener al menos un 25% de batería y al menos un 10% de cobertura:

```
>>> power = 10
>>> signal_4g = 60

>>> power > 25 and signal_4g > 10
False
```

Dado que estamos en un `and` y la primera condición `power > 25` no se cumple, se produce un **cortocircuito** y no se sigue evaluando el resto de la expresión porque ya se sabe que va

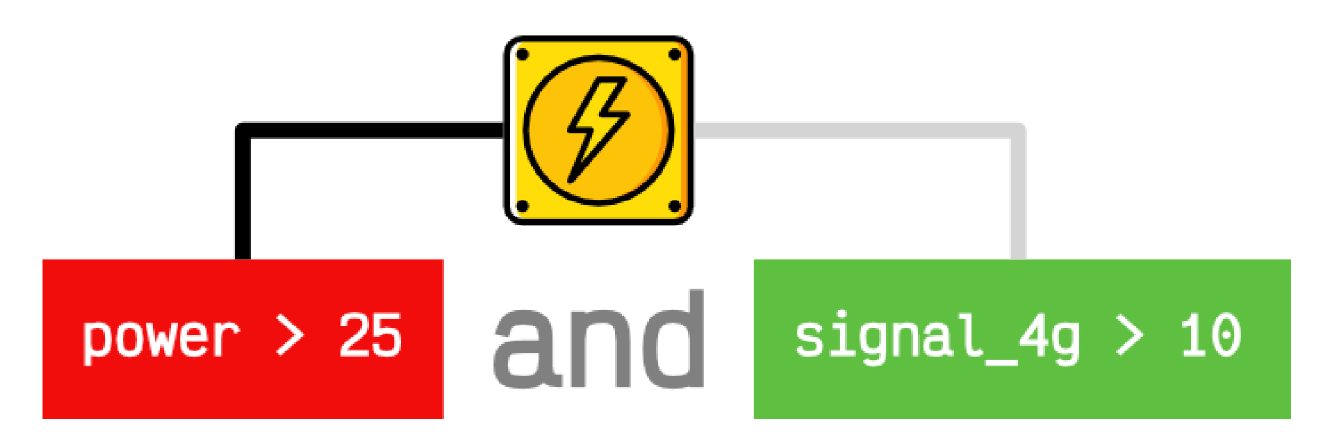



Figura 4: Cortocircuito para expresión lógica «and»

a dar `False`.

Otro ejemplo. Para poder **hacer una llamada VoIP** necesitamos tener al menos un 40% de batería o al menos un 30% de cobertura:

```
>>> power = 50
>>> signal_4g = 20

>>> power > 40 or signal_4g > 30
True
```

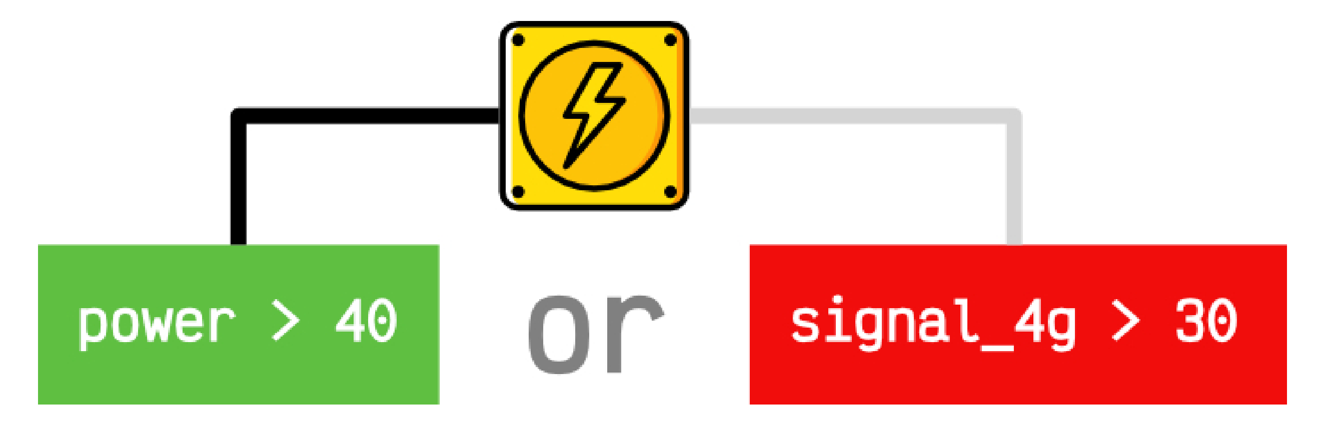



Figura 5: Cortocircuito para expresión lógica «or»

Dado que estamos en un `or` y la primera condición `power > 40` se cumple, se produce un **cortocircuito** y no se sigue evaluando el resto de la expresión porque ya se sabe que va a dar `True`.

---

**Nota:** Si no se produjera un cortocircuito en la evaluación de la expresión, se seguiría comprobando todas las condiciones posteriores hasta llegar al final de la misma.

---

### «Booleanos» en condiciones

Cuando queremos preguntar por la **veracidad** de una determinada variable «booleana» en una condición, la primera aproximación que parece razonable es la siguiente:

```
>>> is_cold = True

>>> if is_cold == True:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Pero podemos *simplificar* esta condición tal que así:

```
>>> if is_cold:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Hemos visto una comparación para un valor «booleano» verdadero (`True`). En el caso de que la comparación fuera para un valor falso lo haríamos así:

```
>>> is_cold = False

>>> if not is_cold:  # Equivalente a if is_cold == False
...     print('Usa camiseta')
... else:
...     print('Coge chaqueta')
...
Usa camiseta
```

De hecho, si lo pensamos, estamos reproduciendo bastante bien el *lenguaje natural*:

- Si hace frío, coge chaqueta.
- Si no hace frío, usa camiseta.

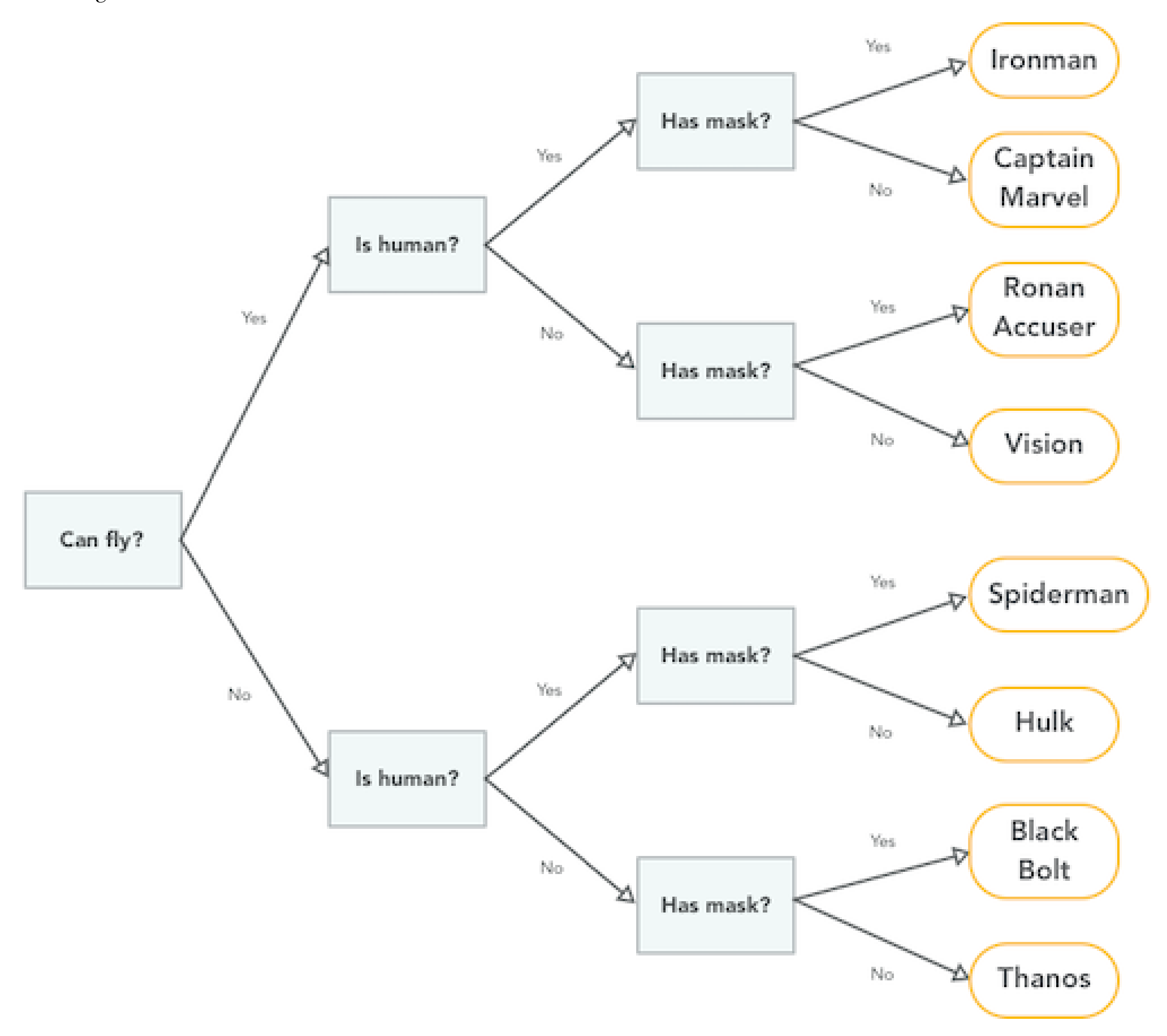
---

**Ejercicio**

Implemente un «clon» de Akinator que permita adivinar un personaje de Marvel en base a las tres preguntas siguientes:

1. ¿Puede volar?
2. ¿Es humano?

3. ¿Tiene máscara?




**Ejemplo**

- Entrada: `can_fly = True`, `is_human = True` y `has_mask = True`
- Salida: `Ironman`

No use `input()` ya que puede complicar la gestión de los valores booleanos.

## Valor nulo

### Nivel intermedio

`None` es un valor especial de Python que almacena el **valor nulo**[4]. Veamos cómo se comporta al incorporarlo en condiciones de veracidad:

```
>>> value = None

>>> if value:
...     print('Value has some useful value')
... else:
...     # value podría contener None, False (u otro)
...     print('Value seems to be void')
...
Value seems to be void
```

Para distinguir `None` de los valores propiamente booleanos, se recomienda el uso del operador `is`. Veamos un ejemplo en el que tratamos de averiguar si un valor **es nulo**:

```
>>> value = None

>>> if value is None:
...     print('Value is clearly None')
... else:
...     # value podría contener True, False (u otro)
...     print('Value has some useful value')
...
Value is clearly None
```

De igual forma, podemos usar esta construcción para el caso contrario. La forma «pitónica» de preguntar si algo **no es nulo** es la siguiente:

```
>>> value = 99

>>> if value is not None:
...     print(f'{value=}')
...
value=99
```

### Nivel avanzado

Cabe preguntarse por qué utilizamos `is` en vez del operador `==` al comprobar si un valor es nulo, ya que ambas aproximaciones nos dan el mismo resultado[7]:

[4] Lo que en otros lenguajes se conoce como `nil`, `null`, `nothing`.
[7] Uso de `is` en comparación de valores nulos explicada aquí por Jared Grubb.

```
>>> value = None

>>> value is None
True

>>> value == None
True
```

La respuesta es que el operador `is` comprueba únicamente si los identificadores (posiciones en memoria) de dos objetos son iguales, mientras que la comparación `==` puede englobar *muchas otras acciones*. De este hecho se deriva que su ejecución sea mucho más rápida y que se eviten «falsos positivos».

Cuando ejecutamos un programa Python existe una serie de objetos precargados en memoria. Uno de ellos es `None`:

```
>>> id(None)
4314501456
```

Cualquier variable que igualemos al valor nulo, únicamente será una referencia al mismo objeto `None` en memoria:

```
>>> value = None

>>> id(value)
4314501456
```

Por lo tanto, ver si un objeto es `None` es simplemente comprobar que su identificador coincida con el de `None`, que es exactamente el cometido de la función `is()`:

```
>>> id(value) == id(None)
True

>>> value is None
True
```

---

**Truco:** Python carga inicialmente en memoria objetos como `True` o `False`, pero también los números enteros que van desde el -5 hasta el 256. Se entiende que tiene que ver con optimizaciones a nivel de rendimiento.

---

### 4.1.8 Veracidad

**Nivel intermedio**

Cuando trabajamos con expresiones que incorporan valores booleanos, se produce una *conversión implícita* que transforma los tipos de datos involucrados a valores `True` o `False`.

Lo primero que debemos entender de cara comprobar la **veracidad** son los valores que **evalúan a falso** o **evalúan a verdadero**.

Veamos las únicas «cosas» que son evaluadas a `False` en Python:

```
>>> bool(False)
False

>>> bool(None)
False

>>> bool(0)
False

>>> bool(0.0)
False

>>> bool('')  # cadena vacía
False

>>> bool([])  # lista vacía
False

>>> bool(())  # tupla vacía
False

>>> bool({})  # diccionario vacío
False

>>> bool(set())  # conjunto vacío
False
```

**Importante:** El resto de objetos son evaluados a `True` en Python.

Veamos algunos ejemplos que son evaluados a `True` en Python:

```
>>> bool('False')
True

```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> bool(' ')
True

>>> bool(1e-10)
True

>>> bool([0])
True

>>> bool('🦆')
True
```

## Asignación lógica

Es posible utilizar *operadores lógicos* en **sentencias de asignación** sacando partido de las tablas de la verdad que funcionan para estos casos.

Veamos un ejemplo de **asignación lógica** utilizando el operador `or`:

```
>>> b = 0
>>> c = 5

>>> a = b or c

>>> a
5
```

En la línea resaltada podemos ver que se está aplicando una **expresión lógica**, por lo tanto se aplica una conversión implícita de los valores enteros a valores «booleanos». En este sentido el valor `0` se **evalúa a falso** y el valor `5` se evalúa a verdadero. Como estamos en un `or` el resultado será verdadero, que en este caso es el valor 5 asignado finalmente a la variable `a`.

Veamos **el mismo ejemplo de antes** pero utilizando el operador `and`:

```
>>> b = 0
>>> c = 5

>>> a = b and c

>>> a
0
```

En este caso, como estamos en un `and` el resultado será falso, por lo que el valor 0 es asignado finalmente a la variable `a`.

## 4.1.9 Sentencia `match-case`

Una de las novedades más esperadas (y quizás controvertidas) de Python 3.10 fue el llamado Structural Pattern Matching que introdujo en el lenguaje una nueva sentencia condicional. Ésta se podría asemejar a la sentencia «switch» que ya existe en otros lenguajes de programación.

### Comparando valores

En su versión más simple, el «pattern matching» permite comparar un valor de entrada con una serie de literales. Algo así como un conjunto de sentencias «if» encadenadas. Veamos esta aproximación mediante un ejemplo:

```
>>> color = '#FF0000'

>>> match color:
...     case '#FF0000':
...         print('🔴')
...     case '#00FF00':
...         print('🟢')
...     case '#0000FF':
...         print('🔵')
...
🔴
```

¿Qué ocurre si el valor que comparamos no existe entre las opciones disponibles? Pues en principio, nada, ya que este caso no está cubierto. Si lo queremos controlar, hay que añadir una nueva regla utilizando el subguión _ como patrón:

```
>>> color = '#AF549B'

>>> match color:
...     case '#FF0000':
...         print('🔴')
...     case '#00FF00':
...         print('🟢')
...     case '#0000FF':
...         print('🔵')
...     case _:
...         print('Unknown color!')
...
Unknown color!
```

**Ejercicio**

Escriba un programa en Python que pida (por separado) dos valores numéricos y un operando (suma, resta, multiplicación, división) y calcule el resultado de la operación, usando para ello la sentencia `match-case`.

Controlar que la operación no sea una de las cuatro predefinidas. En este caso dar un mensaje de error y no mostrar resultado final.

**Ejemplo**

- Entrada: `4`, `3`, `+`
- Salida: `4+3=7`

### Patrones avanzados

**Nivel avanzado**

La sentencia `match-case` va mucho más allá de una simple comparación de valores. Con ella podremos deconstruir estructuras de datos, capturar elementos o mapear valores.

Para ejemplificar varias de sus funcionalidades, vamos a partir de una *tupla* que representará un punto en el plano (2 coordenadas) o en el espacio (3 coordenadas). Lo primero que vamos a hacer es detectar en qué dimensión se encuentra el punto:

```
>>> point = (2, 5)

>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(2,5) is in plane

>>> point = (3, 1, 7)

>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(3,1,7) is in space
```

En cualquier caso, esta aproximación permitiría un punto formado por «strings»:

```
>>> point = ('2', '5')

>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(2,5) is in plane
```

Por lo tanto, en un siguiente paso, podemos restringir nuestros patrones a valores enteros:

```
>>> point = ('2', '5')

>>> match point:
...     case (int(), int()):
...         print(f'{point} is in plane')
...     case (int(), int(), int()):
...         print(f'{point} is in space')
...     case _:
...         print('Unknown!')
...
Unknown!

>>> point = (3, 9, 1)

>>> match point:
...     case (int(), int()):
...         print(f'{point} is in plane')
...     case (int(), int(), int()):
...         print(f'{point} is in space')
...     case _:
...         print('Unknown!')
...
(3, 9, 1) is in space
```

Imaginemos ahora que nos piden calcular la distancia del punto al origen. Debemos tener en cuenta que, a priori, desconocemos si el punto está en el plano o en el espacio:

```
>>> point = (8, 3, 5)

>>> match point:
...     case (int(x), int(y)):
...         dist_to_origin = (x ** 2 + y ** 2) ** (1 / 2)
...     case (int(x), int(y), int(z)):
...         dist_to_origin = (x ** 2 + y ** 2 + z ** 2) ** (1 / 2)
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     case _:
...         print('Unknown!')
...

>>> dist_to_origin
9.899494936611665
```

Con este enfoque, nos aseguramos que los puntos de entrada deben tener todas sus coordenadas como valores enteros:

```
>>> point = ('8', 3, 5)  # Nótese el 8 como "string"

>>> match point:
...     case (int(x), int(y)):
...         dist_to_origin = (x ** 2 + y ** 2) ** (1 / 2)
...     case (int(x), int(y), int(z)):
...         dist_to_origin = (x ** 2 + y ** 2 + z ** 2) ** (1 / 2)
...     case _:
...         print('Unknown!')
...
Unknown!
```

Cambiando de ejemplo, veamos un fragmento de código en el que tenemos que **comprobar la estructura de un bloque de autenticación** definido mediante un *diccionario*. Los métodos válidos de autenticación son únicamente dos: bien usando nombre de usuario y contraseña, o bien usando correo electrónico y «token» de acceso. Además, los valores deben venir en formato cadena de texto:

```
>>> # Lista de diccionarios
>>> auths = [
...     {'username': 'sdelquin', 'password': '1234'},
...     {'email': 'sdelquin@gmail.com', 'token': '4321'},
...     {'email': 'test@test.com', 'password': 'ABCD'},
...     {'username': 'sdelquin', 'password': 1234}
... ]

>>> for auth in auths:
...     print(auth)
...     match auth:
...         case {'username': str(username), 'password': str(password)}:
...             print('Authenticating with username and password')
...             print(f'{username}: {password}')
...         case {'email': str(email), 'token': str(token)}:
...             print('Authenticating with email and token')
...             print(f'{email}: {token}')
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         case _:
...             print('Authenticating method not valid!')
...     print('---')
...
{'username': 'sdelquin', 'password': '1234'}
Authenticating with username and password
sdelquin: 1234
---
{'email': 'sdelquin@gmail.com', 'token': '4321'}
Authenticating with email and token
sdelquin@gmail.com: 4321
---
{'email': 'test@test.com', 'password': 'ABCD'}
Authenticating method not valid!
---
{'username': 'sdelquin', 'password': 1234}
Authenticating method not valid!
---
```

Cambiando de ejemplo, a continuación veremos un código que nos indica si, dada la edad de una persona, puede beber alcohol:

```
1  >>> age = 21
2
3  >>> match age:
4  ...     case 0 | None:
5  ...         print('Not a person')
6  ...     case n if n < 17:
7  ...         print('Nope')
8  ...     case n if n < 22:
9  ...         print('Not in the US')
10 ...     case _:
11 ...         print('Yes')
12 ...
13 Not in the US
```

- En la **línea 4** podemos observar el uso del operador **OR**.
- En las **líneas 6 y 8** podemos observar el uso de condiciones dando lugar a **cláusulas guarda**.

### 4.1.10 Operador morsa

**Nivel avanzado**

A partir de Python 3.8 se incorpora el operador morsa[5] que permite unificar **sentencias de asignación dentro de expresiones**. Su nombre proviene de la forma que adquiere `:=`

Supongamos un ejemplo en el que computamos el perímetro de una circunferencia, indicando al usuario que debe incrementarlo siempre y cuando no llegue a un mínimo establecido.

**Versión tradicional**

```
>>> radius = 4.25
... perimeter = 2 * 3.14 * radius
... if perimeter < 100:
...     print('Increase radius to reach minimum perimeter')
...     print('Actual perimeter: ', perimeter)
...
Increase radius to reach minimum perimeter
Actual perimeter:  26.69
```

**Versión con operador morsa**

```
>>> radius = 4.25
... if (perimeter := 2 * 3.14 * radius) < 100:
...     print('Increase radius to reach minimum perimeter')
...     print('Actual perimeter: ', perimeter)
...
Increase radius to reach minimum perimeter
Actual perimeter:  26.69
```

---

**Consejo:** Como hemos comprobado, el operador morsa permite realizar asignaciones dentro de expresiones, lo que, en muchas ocasiones, permite obtener un código más compacto. Sería conveniente encontrar un equilibrio entre la expresividad y la legibilidad.

---

[5] Se denomina así porque el operador `:=` tiene similitud con los colmillos de una morsa.

## EJERCICIOS DE REPASO

1. **Escriba un programa en Python que acepte la opción de dos jugadoras en** Piedra-Pape

    - Entrada: `person1=piedra; person2=papel`
    - Salida: `Gana persona2: El papel envuelve a la piedra`

2. **Escriba un programa en Python que acepte 3 números y calcule el mínimo (`solución`).**

    - Entrada: `7, 4, 9`
    - Salida: `4`

3. **Escriba un programa en Python que acepte un país (como «string») y muestre por pa**

    - Entrada: `Italia`
    - Salida:

4. **Escriba un programa en Python que acepte 3 códigos de teclas y muestre por pantalla**

    - Entrada: `tecla1=Ctrl; tecla2=Alt; tecla3=Del;`
    - Salida: `Log out`

5. **Escriba un programa en Python que acepte edad, peso, pulso y plaquetas, y determine**

    - Entrada: `edad=34; peso=81; heartbeat=70; plaquetas=150000`
    - Salida: `Apto para donar sangre`

6. pycheck: **red_square**

## EJERCICIOS EXTERNOS

1. Return the day
2. Return negative
3. What's the real floor?
4. Area or Perimeter
5. Check same case
6. Simple multiplication
7. Quarter of the year

8. Grade book
9. Transportation on vacation
10. Safen User Input Part I - htmlspecialchars
11. Remove an exclamation mark from the end of string
12. Pythagorean triple
13. How much water do I need?
14. Set Alarm
15. Compare within margin
16. Will you make it?
17. Plural
18. Student's final grade
19. Drink about
20. Switch it up!
21. Floating point comparison
22. No zeros for heros
23. Tip calculator
24. Grader
25. Evil or Odious
26. Validate code with simple regex
27. Fuel calculator

## AMPLIAR CONOCIMIENTOS

- How to Use the Python or Operator
- Conditional Statements in Python (if/elif/else)

## 4.2 Bucles



Cuando queremos hacer algo más de una vez, necesitamos recurrir a un **bucle**. En esta sección veremos las distintas sentencias en Python que nos permiten repetir un bloque de código.[1]

### 4.2.1 La sentencia `while`

El primer mecanismo que existe en Python para repetir instrucciones es usar la sentencia `while`. La semántica tras esta sentencia es: «Mientras se cumpla la condición haz algo».

Veamos un sencillo bucle que repite un saludo mientras así se desee:

```
>>> want_greet = 'S'  # importante dar un valor inicial

>>> while want_greet == 'S':
...     print('Hola qué tal!')
...     want_greet = input('¿Quiere otro saludo? [S/N] ')
... print('Que tenga un buen día')
Hola qué tal!
¿Quiere otro saludo? [S/N] S
Hola qué tal!
¿Quiere otro saludo? [S/N] S
```

(continué en la próxima página)

[1] Foto original de portada por Gary Lopater en Unsplash.

(proviene de la página anterior)

```
Hola qué tal!
¿Quiere otro saludo? [S/N] N
Que tenga un buen día
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/Zwwr8fub

La condición del bucle se comprueba en cada nueva repetición. En este caso chequeamos que la variable `want_greet` sea igual a `'S'`. Dentro del cuerpo del bucle estamos mostrando un mensaje y pidiendo la opción al usuario.

### Romper un bucle while

Python ofrece la posibilidad de *romper* o finalizar un bucle *antes de que se cumpla la condición de parada*.

Supongamos que en el ejemplo anterior, establecemos un máximo de 4 saludos:

```
>>> MAX_GREETS = 4

>>> num_greets = 0
>>> want_greet = 'S'

>>> while want_greet == 'S':
...     print('Hola qué tal!')
...     num_greets += 1
...     if num_greets == MAX_GREETS:
...         print('Máximo número de saludos alcanzado')
...         break
...     want_greet = input('¿Quiere otro saludo? [S/N] ')
... print('Que tenga un buen día')
Hola qué tal!
¿Quiere otro saludo? [S/N] S
Hola qué tal!
¿Quiere otro saludo? [S/N] S
Hola qué tal!
¿Quiere otro saludo? [S/N] S
Hola qué tal!
Máximo número de saludos alcanzado
Que tenga un buen día
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/0wwtyaDF

Como hemos visto en este ejemplo, `break` nos permite finalizar el bucle una vez que hemos llegado al máximo número de saludos. Pero si no hubiéramos llegado a dicho límite, el bucle habría seguido hasta que el usuario indicara que no quiere más saludos.

Otra forma de resolver este ejercicio sería incorporar una condición al bucle:

```
while want_greet == 'S' and num_questions < MAX_GREETS:
    ...
```

### Comprobar la rotura

#### Nivel intermedio

Python nos ofrece la posibilidad de **detectar si el bucle ha acabado de forma ordinaria**, esto es, ha finalizado por no cumplirse la condición establecida. Para ello podemos hacer uso de la sentencia `else` como parte del propio bucle. Si el bucle while finaliza normalmente (sin llamada a `break`) el flujo de control pasa a la sentencia opcional `else`.

Veamos su comportamiento siguiendo con el ejemplo que venimos trabajando:

```
>>> MAX_GREETS = 4

>>> num_greets = 0
>>> want_greet = 'S'

>>> while want_greet == 'S':
...     print('Hola qué tal!')
...     num_greets += 1
...     if num_greets == MAX_GREETS:
...         print('Máximo número de saludos alcanzado')
...         break
...     want_greet = input('¿Quiere otro saludo? [S/N] ')
... else:
...     print('Usted no quiere más saludos')
... print('Que tenga un buen día')
Hola qué tal!
¿Quiere otro saludo? [S/N] S
Hola qué tal!
¿Quiere otro saludo? [S/N] N
Usted no quiere más saludos
Que tenga un buen día
```

**Importante:** Si hubiéramos agotado el número de saludos NO se habría ejecutado la cláusula `else` del bucle ya que habríamos roto el flujo con un `break`.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/jwwtpivu

## Continuar un bucle

### Nivel intermedio

Hay situaciones en las que, en vez de romper un bucle, nos interesa **saltar adelante hacia la siguiente repetición**. Para ello Python nos ofrece la sentencia `continue` que hace precisamente eso, descartar el resto del código del bucle y saltar a la siguiente iteración.

Continuamos con el ejemplo anterior y vamos a contar el número de respuestas válidas:

```
>>> want_greet = 'S'
>>> valid_options = 0

>>> while want_greet == 'S':
...     print('Hola qué tal!')
...     want_greet = input('¿Quiere otro saludo? [S/N] ')
...     if want_greet not in 'SN':
...         print('No le he entendido pero le saludo')
...         want_greet = 'S'
...         continue
...     valid_options += 1
... print(f'{valid_options} respuestas válidas')
... print('Que tenga un buen día')
Hola qué tal!
¿Quiere otro saludo? [S/N] S
Hola qué tal!
¿Quiere otro saludo? [S/N] A
No le he entendido pero le saludo
Hola qué tal!
¿Quiere otro saludo? [S/N] B
No le he entendido pero le saludo
Hola qué tal!
¿Quiere otro saludo? [S/N] N
2 respuestas válidas
Que tenga un buen día
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/YwwtlAp8

### Bucle infinito

Si no establecemos correctamente la **condición de parada** o bien el valor de alguna variable está fuera de control, es posible que lleguemos a una situación de bucle infinito, del que nunca podamos salir. Veamos un ejemplo de esto:

```
>>> num = 1

>>> while num != 10:
...     num += 2
...
# CTRL-C
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

El problema que surje es que la variable `num` toma los valores `1, 3, 5, 7, 9, 11, ...` por lo que nunca se cumple la condición de parada del bucle. Esto hace que repitamos «eternamente» la instrucción de incremento.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/AfrZroa

Una posible solución a este error es reescribir la condición de parada en el bucle:

```
>>> num = 1

>>> while num < 10:
...     num += 2
...
```

---

**Truco:** Para abortar una situación de *bucle infinito* podemos pulsar en el teclado la combinación `CTRL-C`. Se puede ver reflejado en el intérprete de Python por `KeyboardInterrupt`.

---

Hay veces que un **supuesto bucle «infinito»** puede ayudarnos a resolver un problema. Imaginemos que queremos escribir un programa que ayude al profesorado a introducir las notas de un examen. Si la nota no está en el intervalo $[0, 10]$ mostramos un mensaje de error, en otro caso seguimos pidiendo valores:

```
>>> while True:
...     mark = float(input('Introduzca nueva nota: '))
...     if not(0 <= mark <= 10):
...         print('Nota fuera de rango')
...         break
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     print(mark)
...
Introduzca nueva nota: 5
5.0
Introduzca nueva nota: 3
3.0
Introduzca nueva nota: 11
Nota fuera de rango
```

El código anterior se podría enfocar haciendo uso del *operador morsa*:

```
>>> while 0 <= (mark := float(input('Introduzca una nueva nota: '))) <= 10:
...     print(mark)
... print('Nota fuera de rango')
Introduzca una nueva nota: 5
5.0
Introduzca una nueva nota: 3
3.0
Introduzca una nueva nota: 11
Nota fuera de rango
```

**Ejercicio**

Escriba un programa que encuentre todos los múltiplos de 5 menores que un valor dado:

**Ejemplo**

- Entrada: `36`
- Salida: `5 10 15 20 25 30 35`

### 4.2.2 La sentencia `for`

Python permite recorrer aquellos tipos de datos que sean **iterables**, es decir, que admitan *iterar*[2] sobre ellos. Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (*recorridas*) son: cadenas de texto, listas, diccionarios, ficheros, etc. La sentencia `for` nos permite realizar esta acción.

A continuación se plantea un ejemplo en el que vamos a recorrer (iterar) una cadena de texto:

[2] Realizar cierta acción varias veces. En este caso la acción es tomar cada elemento.

```
>>> word = 'Python'

>>> for letter in word:
...     print(letter)
...
P
y
t
h
o
n
```

La clave aquí está en darse cuenta que el bucle va tomando, en cada iteración, cada uno de los elementos de la variable que especifiquemos. En este caso concreto `letter` va tomando cada una de las letras que existen en `word`, porque una cadena de texto está formada por elementos que son caracteres.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/Pft6R2e

---

**Importante:** La variable que utilizamos en el bucle `for` para ir tomando los valores puede tener **cualquier nombre**. Al fin y al cabo es una variable que definimos según nuestras necesidades. Tener en cuenta que se suele usar un nombre en singular.

---

### Romper un bucle for

Una sentencia break dentro de un `for` rompe el bucle, *igual que veíamos* para los bucles `while`. Veamos un ejemplo con el código anterior. En este caso vamos a recorrer una cadena de texto y pararemos el bucle cuando encontremos una letra *t* minúscula:

```
>>> word = 'Python'

>>> for letter in word:
...     if letter == 't':
...         break
...     print(letter)
...
P
y
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/zfyqkbJ

**Truco:** Tanto la *comprobación de rotura de un bucle* como la *continuación a la siguiente iteración* se llevan a cabo del mismo modo que hemos visto con los bucles de tipo `while`.

**Ejercicio**

Dada una cadena de texto, indique el número de vocales que tiene.

**Ejemplo**

- Entrada: `Supercalifragilisticoespialidoso`
- Salida: `15`

### Secuencias de números

Es muy habitual hacer uso de secuencias de números en bucles. Python no tiene una instrucción específica para ello. Lo que sí aporta es una función `range()` que devuelve un *flujo de números* en el rango especificado. Una de las grandes ventajas es que la «lista» generada no se construye explícitamente, sino que cada valor se genera bajo demanda. Esta técnica mejora el consumo de recursos, especialmente en términos de memoria.

La técnica para la generación de secuencias de números es muy similar a la utilizada en los *«slices»* de cadenas de texto. En este caso disponemos de la función `range(start, stop, step)`:

- **start**: Es *opcional* y tiene valor por defecto **0**.
- **stop**: es *obligatorio* (siempre se llega a 1 menos que este valor).
- **step**: es *opcional* y tiene valor por defecto **1**.

`range()` devuelve un *objeto iterable*, así que iremos obteniendo los valores paso a paso con una sentencia `for ... in`[3]. Veamos diferentes ejemplos de uso:

**Rango:** $[0, 1, 2]$

```
>>> for i in range(0, 3):
...     print(i)
...
0
1
2

>>> for i in range(3):  # No hace falta indicar el inicio si es 0
```

(continué en la próxima página)

[3] O convertir el objeto a una secuencia como una lista.

(proviene de la página anterior)

```
...     print(i)
...
0
1
2
```

**Rango:** $[1, 3, 5]$

```
>>> for i in range(1, 6, 2):
...     print(i)
...
1
3
5
```

**Rango:** $[2, 1, 0]$

```
>>> for i in range(2, -1, -1):
...     print(i)
...
2
1
0
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/vfywE45

**Truco:** Se suelen utilizar nombres de variables `i`, `j`, `k` para lo que se denominan **contadores**. Este tipo de variables toman valores numéricos enteros como en los ejemplos anteriores. No conviene generalizar el uso de estas variables a situaciones en las que, claramente, tenemos la posibilidad de asignar un nombre semánticamente más significativo. Esto viene de tiempos antiguos en FORTRAN donde `i` era la primera letra que tenía valor entero por defecto.

**Ejercicio**

Determine si un número dado es un número primo.

*No es necesario implementar ningún algoritmo en concreto. La idea es probar los números menores al dado e ir viendo si las divisiones tienen resto cero o no.*

¿Podría optimizar su código? ¿Realmente es necesario probar con tantos divisores?

**Ejemplo**

- Entrada: `11`

- Salida: `Es primo`

### Usando el guión bajo

**Nivel avanzado**

Hay situaciones en las que **no necesitamos usar la variable** que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces.

Para estos casos se suele recomendar usar el **guión bajo _** como **nombre de variable**, que da a entender que no estamos usando esta variable de forma explícita:

```
>>> for _ in range(10):
...     print('Repeat me 10 times!')
...
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
```

**Ejercicio**

Imprima los 100 primeros números de la sucesión de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

## 4.2.3 Bucles anidados

Como ya vimos en las *sentencias condicionales*, el *anidamiento* es una técnica por la que incluimos distintos niveles de encapsulamiento de sentencias, unas dentro de otras, con mayor nivel de profundidad. En el caso de los bucles también es posible hacer anidamiento.

Veamos un ejemplo de 2 bucles anidados en el que generamos todas las tablas de multiplicar:

```
>>> for num_table in range(1, 10):
...     for mul_factor in range(1, 10):
```

(continué en la próxima página)

[4] Foto de Matrioskas por *Marina Yufereva*‘_‘ en Escáner Cultural.



Figura 6: Muñecas rusas Matrioskas para ejemplificar el anidamiento[4]

(proviene de la página anterior)

```
...         result = num_table * mul_factor
...         print(f'{num_table} * {mul_factor} = {result}')
...
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
```

(continué en la próxima página)

(proviene de la página anterior)

```
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
6 x 4 = 24
6 x 5 = 30
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 54
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
```

Lo que está ocurriendo en este código es que, para cada valor que toma la variable `i`, la otra variable `j` toma todos sus valores. Como resultado tenemos una combinación completa de los valores en el rango especificado.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/pwwtctK6

**Nota:**

- Podemos añadir todos los niveles de anidamiento que queramos. Eso sí, hay que tener en cuenta que cada nuevo nivel de anidamiento supone un importante aumento de la complejidad ciclomática de nuestro código, lo que se traduce en mayores tiempos de ejecución.
- Los bucles anidados también se pueden aplicar en la sentencia `while`.

**Ejercicio**

Dado su tamaño, muestre por pantalla un mosaico donde la diagonal principal esté representada por `X`, la parte inferior por `D` y la parte superior por `U`.

**Ejemplo**

- Entrada: `5`
- Salida:

```
X U U U U
D X U U U
D D X U U
D D D X U
D D D D X
```

## EJERCICIOS DE REPASO

1. **Escriba un programa que encuentre la mínima secuencia de múltiplos de 3 (distintos)**

   - Entrada: `45`
   - Salida: `0, 3, 6, 9, 12, 15`

2. Escriba un programa que pida nombre y apellidos de una persona (usando un solo `input`) y repita la pregunta mientras el nombre no esté en formato título (`solución`).

```
¿Su nombre? ana torres blanco
Error. Debe escribirlo correctamente
¿Su nombre? Ana torres blanco
Error. Debe escribirlo correctamente
¿Su nombre? Ana Torres blanco
```

(continué en la próxima página)

(proviene de la página anterior)

```
Error. Debe escribirlo correctamente
¿Su nombre? Ana Torres Blanco
```

3. Escriba un programa en Python que realice las siguientes 9 multiplicaciones. ¿Nota algo raro en el resultado? (`solución`)

$$1 \cdot 1$$
$$11 \cdot 11$$
$$111 \cdot 111$$
$$\vdots$$
$$111111111 \cdot 111111111$$

4. **Escriba un programa en Python que acepte una cadena de texto e indique si todos sus**

   - Entrada: `hello-world`
   - Salida: `Se han encontrado caracteres no alfabéticos`

5. Escriba un programa en Python que acepte un número entero $n$ y realice el siguiente cálculo de productos sucesivos (`solución`):

$$\prod_{i=1}^{n} i^2 = 1^2 \cdot 2^2 \cdot 3^2 \cdot \cdots \cdot n^2$$

6. **Escriba un programa en Python que acepte dos cadenas de texto y compute el produc**

   - Entrada: `str1=abc; str2=123`
   - Salida: `a1 a2 a3 b1 b2 b3 c1 c2 c3`

7. **Escriba un programa en Python que acepte dos valores enteros ($x$ e $y$) que representan**

   - Entrada: `objetivo_x=7; objetivo_y=8;`
   - Salida: `(0, 0) (1, 2) (3, 3) (4, 5) (6, 6) (7, 8)`

8. **Escriba un programa que calcule la distancia hamming entre dos *cadenas de texto* de l**

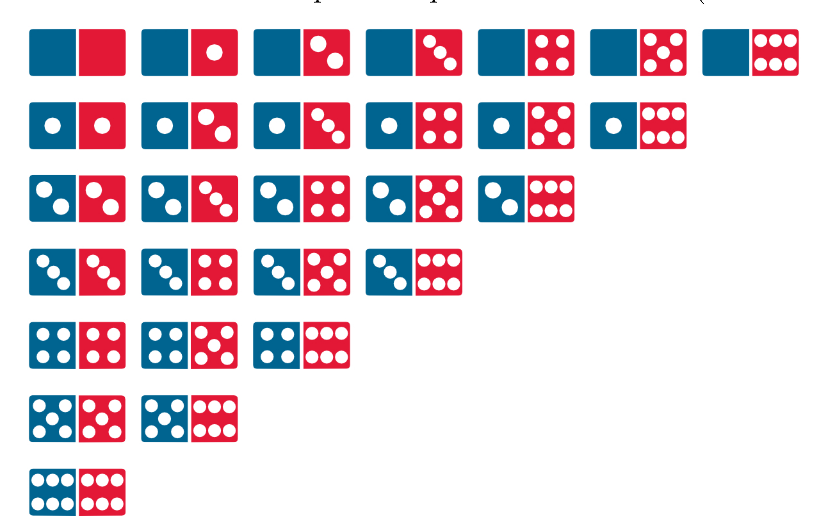
   - Entrada: `0001010011101` y `0000110010001`
   - Salida: `4`

9. **Escriba un programa que calcule el máximo común divisor entre dos números enteros.**

   - Entrada: `a=12; b=44`

   - Salida: `4`

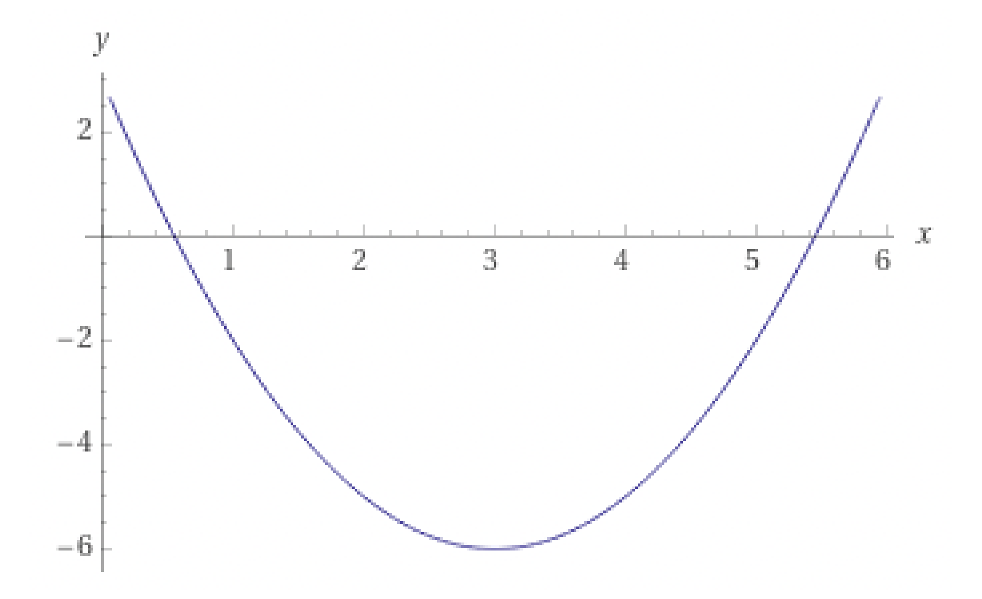
10. Escriba un programa que muestre por pantalla todas las fichas del dominó. La ficha «en blanco» se puede representar con un 0 (`solución`).



```
0|0 0|1 0|2 0|3 0|4 0|5 0|6
1|1 1|2 1|3 1|4 1|5 1|6
2|2 2|3 2|4 2|5 2|6
3|3 3|4 3|5 3|6
4|4 4|5 4|6
5|5 5|6
6|6
```

11. Escriba un programa que calcule el valor de $x$ para el que la función $f(x) = x^2 - 6x + 3$ obtiene su menor resultado. Centre la búsqueda en el rango $[-9, 9]$ sólo con valores enteros (`solución`).

El resultado es: $x = 3$ y $f(x) = -6$




12. Escriba un programa que muestre (por filas) la Tabla ASCII, empezando con el código 33 y terminando con el 127 (`solución`):

```
033 !   034 "   035 #   036 $   037 %
038 &   039 '   040 (   041 )   042 *
043 +   044 ,   045 -   046 .   047 /
048 0   049 1   050 2   051 3   052 4
053 5   054 6   055 7   056 8   057 9
058 :   059 ;   060 <   061 =   062 >
063 ?   064 @   065 A   066 B   067 C
068 D   069 E   070 F   071 G   072 H
073 I   074 J   075 K   076 L   077 M
078 N   079 O   080 P   081 Q   082 R
083 S   084 T   085 U   086 V   087 W
088 X   089 Y   090 Z   091 [   092 \
093 ]   094 ^   095 _   096 `   097 a
098 b   099 c   100 d   101 e   102 f
103 g   104 h   105 i   106 j   107 k
108 l   109 m   110 n   111 o   112 p
113 q   114 r   115 s   116 t   117 u
118 v   119 w   120 x   121 y   122 z
123 {   124 |   125 }   126 ~   127
```

13. Escriba un programa que permita al usuario adivinar un número. Indicar si el número buscado es menor o mayor que el que se está preguntando y mostrar igualmente el número de intentos hasta encontrar el número objetivo (`solución`):

```
Introduzca número: 50
Mayor
Introduzca número: 100
```

(continué en la próxima página)

(proviene de la página anterior)

```
Menor
Introduzca número: 90
Menor
Introduzca número: 87
✅ ¡Enhorabuena! Has encontrado el número en 4 intentos
```

14. pycheck: **tennis_game**
15. pycheck: **tennis_set**

## EJERCICIOS EXTERNOS

1. Summation
2. Find nearest square number
3. Bin to decimal
4. altERnaTIng cAsE
5. Fake binary
6. Correct the mistakes of the character recognition software
7. String cleaning
8. Sum of multiples
9. ASCII Total
10. Collatz Conjecture (3n+1)

## AMPLIAR CONOCIMIENTOS

- The Python range() Function
- How to Write Pythonic Loops
- For Loops in Python (Definite Iteration)
- Python «while» Loops (Indefinite Iteration)

CAPÍTULO 5

# Estructuras de datos

Si bien ya hemos visto una sección sobre *Tipos de datos*, podríamos hablar de tipos de datos más complejos en Python que se constituyen en **estructuras de datos**. Si pensamos en estos elementos como *átomos*, las estructuras de datos que vamos a ver sería *moléculas*. Es decir, combinamos los tipos básicos de formas más complejas. De hecho, esta distinción se hace en el Tutorial oficial de Python. Trataremos distintas estructuras de datos como listas, tuplas, diccionarios y conjuntos.

## 5.1 Listas



Las listas permiten **almacenar objetos** mediante un **orden definido** y con posibilidad de duplicados. Las listas son estructuras de datos **mutables**, lo que significa que podemos añadir, eliminar o modificar sus elementos.[1]

### 5.1.1 Creando listas

Una lista está compuesta por *cero* o *más elementos*. En Python debemos escribir estos elementos separados por *comas* y dentro de *corchetes*. Veamos algunos ejemplos de listas:

```
>>> empty_list = []

>>> languages = ['Python', 'Ruby', 'Javascript']

>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]

>>> data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718, (28.2933947, -16.
↪5226597)]
```

**Nota:** Una lista puede contener tipos de **datos heterogéneos**, lo que la hace una estructura

[1] Foto original de portada por Mike Arney en Unsplash.

de datos muy versátil.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/Ofiiare

**Advertencia:** Aunque está permitido, **NUNCA** llames `list` a una variable porque destruirías la función que nos permite crear listas. Y tampoco uses nombres derivados como `_list` o `list_` ya que no son nombres representativos que *identifiquen el propósito de la variable*.

**Ejercicio**

Cree una lista con las 5 ciudades que más le gusten.

### 5.1.2 Conversión

Para convertir otros tipos de datos en una lista podemos usar la función `list()`:

```
>>> # conversión desde una cadena de texto
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
```

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto `Python` se ha creado una lista con *6* elementos, donde cada uno de ellos representa un carácter de la cadena. Podemos *extender* este comportamiento a cualquier otro tipo de datos que permita ser iterado (*iterables*).

Otro ejemplo interesante de conversión puede ser la de los *rangos*. En este caso queremos obtener una **lista explícita** con los valores que constituyen el rango $[0, 9]$:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### Lista vacía

Existe una manera particular de usar `list()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en una lista, con lo que obtendremos una *lista vacía*:

```
>>> list()
[]
```

**Truco:** Para crear una lista vacía, se suele recomendar el uso de `[]` frente a `list()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

## 5.1.3 Operaciones con listas

### Obtener un elemento

Igual que en el caso de las *cadenas de texto*, podemos obtener un elemento de una lista a través del **índice** (lugar) que ocupa. Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[1]
'Huevos'

>>> shopping[2]
'Aceite'

>>> shopping[-1]  # acceso con índice negativo
'Aceite'
```

El *índice* que usemos para acceder a los elementos de una lista tiene que estar comprendido entre los límites de la misma. Si usamos un índice antes del comienzo o después del final obtendremos un error (*excepción*):

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continué en la próxima página)

(proviene de la página anterior)

```
IndexError: list index out of range

>>> shopping[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

### Trocear una lista

El troceado de listas funciona de manera totalmente análoga al *troceado de cadenas*. Veamos algunos ejemplos:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[0:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[2:4]
['Aceite', 'Sal']

>>> shopping[-1:-4:-1]
['Limón', 'Sal', 'Aceite']

>>> # Equivale a invertir la lista
>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

En el troceado de listas, a diferencia de lo que ocurre al obtener elementos, no debemos preocuparnos por acceder a *índices inválidos* (fuera de rango) ya que Python los restringirá a los límites de la lista:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[10:]
[]

>>> shopping[-100:2]
['Agua', 'Huevos']

```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> shopping[2:100]
['Aceite', 'Sal', 'Limón']
```

**Importante:** Ninguna de las operaciones anteriores modifican la lista original, simplemente devuelven una lista nueva.

### Invertir una lista

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

**Conservando la lista original:** *Opción 1*: Mediante *troceado* de listas con *step* negativo:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

*Opción 2*: Mediante la función `reversed()`:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> list(reversed(shopping))
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

**Modificando la lista original:** Utilizando la función `reverse()` (nótese que es sin *«d»* al final):

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.reverse()

>>> shopping
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

### Añadir al final de la lista

Una de las operaciones más utilizadas en listas es añadir elementos al final de las mismas. Para ello Python nos ofrece la función `append()`. Se trata de un método «destructivo» que modifica la lista original:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.append('Atún')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Atún']
```

### Creando desde vacío

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. Se podría hablar de un **patrón creación**.

Supongamos un ejemplo en el que queremos construir una lista con los números pares del $[0, 20)$:

```
>>> even_numbers = []

>>> for i in range(20):
...     if i % 2 == 0:
...         even_numbers.append(i)
...

>>> even_numbers
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/2fiS9Ax

### Añadir en cualquier posición de una lista

Ya hemos visto cómo añadir elementos al final de una lista. Sin embargo, Python ofrece una función `insert()` que vendría a ser una generalización de la anterior, para incorporar elementos en cualquier posición. Simplemente debemos especificar el índice de inserción y el elemento en cuestión. También se trata de una función *destructiva*[2]:

[2] Cuando hablamos de que una función/método es «destructiva/o» significa que modifica la lista (objeto) original, no que la destruye.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(1, 'Jamón')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Aceite']

>>> shopping.insert(3, 'Queso')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Queso', 'Aceite']
```

**Nota:** El índice que especificamos en la función `insert()` lo podemos intepretar como la posición *delante* (a la izquierda) de la cual vamos a colocar el nuevo valor en la lista.

Al igual que ocurría con el *troceado de listas*, en este tipo de inserciones no obtendremos un error si especificamos índices fuera de los límites de la lista. Estos se ajustarán al principio o al final en función del valor que indiquemos:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(100, 'Mermelada')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Mermelada']

>>> shopping.insert(-100, 'Arroz')

>>> shopping
['Arroz', 'Agua', 'Huevos', 'Aceite', 'Mermelada']
```

**Consejo:** Aunque es posible utilizar `insert()` para añadir **elementos al final de una lista**, siempre se recomienda usar `append()` por su mayor legibilidad:

```
>>> values = [1, 2, 3]
>>> values.append(4)
>>> values
[1, 2, 3, 4]

>>> values = [1, 2, 3]
>>> values.insert(len(values), 4)  # don't do it!
>>> values
[1, 2, 3, 4]
```

### Repetir elementos

Al igual que con las *cadenas de texto*, el operador `*` nos permite repetir los elementos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping * 3
['Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite']
```

### Combinar listas

Python nos ofrece dos aproximaciones para combinar listas:

**Conservando la lista original:** Mediante el operador `+` o `+=`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping + fruitshop
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

**Modificando la lista original:** Mediante la función `extend()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.extend(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

Hay que tener en cuenta que `extend()` funciona adecuadamente si pasamos una **lista como argumento**. En otro caso, quizás los resultados no sean los esperados. Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.extend('Limón')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'L', 'i', 'm', 'ó', 'n']
```

El motivo es que `extend()` «recorre» (o itera) sobre cada uno de los elementos del objeto en cuestión. En el caso anterior, al ser una cadena de texto, está formada por caracteres. De ahí el resultado que obtenemos.

Se podría pensar en el uso de `append()` para combinar listas. La realidad es que no funciona exactamente como esperamos; la segunda lista se añadiría como una *sublista* de la principal:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.append(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', ['Naranja', 'Manzana', 'Piña']]
```

### Modificar una lista

Del mismo modo que se *accede a un elemento* utilizando su índice, también podemos modificarlo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[0] = 'Jugo'

>>> shopping
['Jugo', 'Huevos', 'Aceite']
```

En el caso de acceder a un *índice no válido* de la lista, incluso para modificar, obtendremos un error:

```
>>> shopping[100] = 'Chocolate'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

### Modificar con troceado

No sólo es posible modificar un elemento de cada vez, sino que podemos asignar valores a trozos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4]
['Huevos', 'Aceite', 'Sal']

>>> shopping[1:4] = ['Atún', 'Pasta']

>>> shopping
['Agua', 'Atún', 'Pasta', 'Limón']
```

**Nota:** La lista que asignamos no necesariamente debe tener la misma longitud que el trozo que sustituimos.

### Borrar elementos

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

**Por su índice:** Mediante la sentencia `del`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> del shopping[3]

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

**Por su valor:** Mediante la función `remove()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.remove('Sal')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

**Advertencia:** Si existen valores duplicados, la función `remove()` sólo borrará la primera ocurrencia.

**Por su índice (con extracción):** La sentencia `del` y la función `remove()` efectivamente borran el elemento indicado de la lista, pero no «devuelven»[3] nada. Sin embargo, Python nos ofrece la función `pop()` que además de borrar, nos «recupera» el elemento; algo así como una *extracción*. Lo podemos ver como una combinación de *acceso* + *borrado*:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> product = shopping.pop()  # shopping.pop(-1)
>>> product
'Limón'

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal']

>>> product = shopping.pop(2)
>>> product
'Aceite'

>>> shopping
['Agua', 'Huevos', 'Sal']
```

**Nota:** Si usamos la función `pop()` sin pasarle ningún argumento, por defecto usará el índice *-1*, es decir, el último elemento de la lista. Pero también podemos indicarle el índice del elemento a extraer.

**Por su rango:** Mediante troceado de listas:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4] = []

>>> shopping
['Agua', 'Limón']
```

[3] Más adelante veremos el comportamiento de las funciones. Devolver o retornar un valor es el resultado de aplicar una función.

### Borrado completo de la lista

Python nos ofrece, al menos, dos formas para borrar una lista por completo:

1. Utilizando la función `clear()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.clear()  # Borrado in-situ

>>> shopping
[]
```

2. «Reinicializando» la lista a vacío con `[]`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping = []  # Nueva zona de memoria

>>> shopping
[]
```

**Nivel avanzado**

La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> id(shopping)
4416018560
>>> shopping.clear()
>>> id(shopping)  # se mantiene la misma "posición de memoria"
4416018560

>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> id(shopping)
4458688576
>>> shopping = []
>>> id(shopping)  # se crea una nueva "posición de memoria"
4458851520
```

**Ver también:**

La memoria que queda «en el limbo» después de asignar un nuevo valor a la lista es detectada por el **recolector de basura** de Python, quien se encarga de liberar aquellos datos que no están referenciados por ninguna variable.

### Encontrar un elemento

Si queremos descubrir el **índice** que corresponde a un determinado valor dentro la lista podemos usar la función `index()` para ello:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.index('Huevos')
1
```

Tener en cuenta que si el elemento que buscamos no está en la lista, obtendremos un error:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.index('Pollo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'Pollo' is not in list
```

**Nota:** Si buscamos un valor que existe más de una vez en una lista, la función `index()` sólo nos devolverá el índice de la primera ocurrencia.

**Advertencia:** En listas no disponemos de la función `find()` que sí estaba disponible para *cadenas de texto*.

### Pertenencia de un elemento

Si queremos comprobar la existencia de un determinado elemento en una lista, podríamos buscar su índice, pero la forma **pitónica** de hacerlo es utilizar el operador `in`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> 'Aceite' in shopping
True

>>> 'Pollo' in shopping
False
```

**Nota:** El operador `in` siempre devuelve un valor booleano, es decir, verdadero o falso.

**Ejercicio**

Determine si una cadena de texto dada es un **isograma**, es decir, no se repite ninguna letra.

Ejemplos válidos de isogramas:

- *lumberjacks*
- *background*
- *downstream*
- *six-year-old*

### Número de ocurrencias

Para contar cuántas veces aparece un determinado valor dentro de una lista podemos usar la función `count()`:

```
>>> sheldon_greeting = ['Penny', 'Penny', 'Penny']

>>> sheldon_greeting.count('Howard')
0

>>> sheldon_greeting.count('Penny')
3
```

### Convertir lista a cadena de texto

Dada una lista, podemos convetirla a una cadena de texto, uniendo todos sus elementos mediante algún **separador**. Para ello hacemos uso de la función `join()` con la siguiente estructura:




Figura 1: Estructura de llamada a la función `join()`

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> ','.join(shopping)
'Agua,Huevos,Aceite,Sal,Limón'

>>> ' '.join(shopping)
'Agua Huevos Aceite Sal Limón'

>>> '|'.join(shopping)
'Agua|Huevos|Aceite|Sal|Limón'
```

Hay que tener en cuenta que `join()` sólo funciona si *todos sus elementos son cadenas de texto*:

```
>>> ', '.join([1, 2, 3, 4, 5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

**Truco:** Esta función `join()` es realmente la **opuesta** a la de `split()` para *dividir una cadena*.

**Ejercicio**

Consiga la siguiente transformación:

`12/31/20` ➡️ `31-12-2020`

## Ordenar una lista

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

**Conservando lista original:** Mediante la función `sorted()` que devuelve una nueva lista ordenada:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> sorted(shopping)
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

**Modificando la lista original:** Mediante la función `sort()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.sort()

>>> shopping
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

**Ambos métodos** admiten un *parámetro* «booleano» `reverse` para indicar si queremos que la ordenación se haga en **sentido inverso**:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> sorted(shopping, reverse=True)
['Sal', 'Limón', 'Huevos', 'Agua', 'Aceite']
```

### Longitud de una lista

Podemos conocer el número de elementos que tiene una lista con la función `len()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> len(shopping)
5
```

### Iterar sobre una lista

Al igual que *hemos visto con las cadenas de texto*, también podemos *iterar* sobre los elementos de una lista utilizando la sentencia `for`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> for product in shopping:
...     print(product)
...
Agua
Huevos
Aceite
Sal
Limón
```

**Nota:** También es posible usar la sentencia `break` en este tipo de bucles para abortar su ejecución en algún momento que nos interese.

**Ejercicio**

pycheck: **chars_list**

### Iterar usando enumeración

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos **saber su índice** dentro de la misma. Para ello Python nos ofrece la función `enumerate()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> for i, product in enumerate(shopping):
...     print(i, product)
...
0 Agua
1 Huevos
2 Aceite
3 Sal
4 Limón
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/TfiuIZ0

**Truco:** Es posible utilizar el parámetro `start` con `enumerate()` para indicar el índice en el que queremos comenzar. Por defecto es 0.

### Iterar sobre múltiples listas

Python ofrece la posibilidad de iterar sobre **múltiples listas en paralelo** utilizando la función `zip()`. Se basa en ir «juntando» ambas listas elemento a elemento:

Veamos un ejemplo en el que añadimos ciertos detalles a nuestra lista de la compra:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']

>>> for product, detail in zip(shopping, details):
...     print(product, detail)
...
```

(continué en la próxima página)

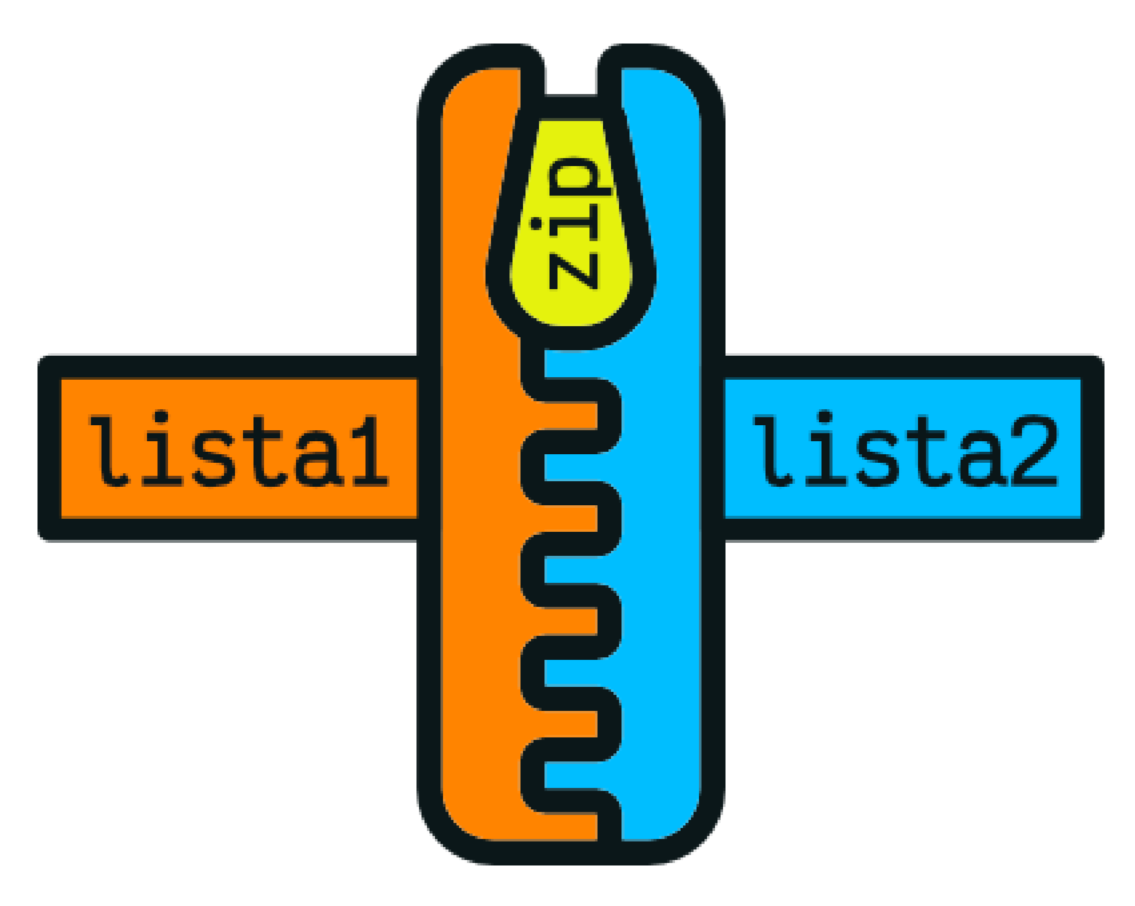



Figura 2: Funcionamiento de `zip()`

(proviene de la página anterior)

```
Agua mineral natural
Aceite de oliva virgen
Arroz basmati
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/lfioilG

**Nota:** En el caso de que las listas no tengan la misma longitud, la función `zip()` realiza la combinación hasta que se agota la lista más corta.

Dado que `zip()` produce un *iterador*, si queremos obtener una **lista explícita** con la combinación en paralelo de las listas, debemos construir dicha lista de la siguiente manera:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']

>>> list(zip(shopping, details))
[('Agua', 'mineral natural'),
 ('Aceite', 'de oliva virgen'),
 ('Arroz', 'basmati')]
```

### Ejercicio

Dados dos vectores (listas) de la misma dimensión, utilice la función `zip()` para calcular su producto escalar.

### Ejemplo

- Entrada:

```
v1 = [4, 3, 8, 1]
v2 = [9, 2, 7, 3]
```

- Salida: `101`

$$v1 \cdot v2 = [4 \cdot 9 + 3 \cdot 2 + 8 \cdot 7 + 1 \cdot 3] = 101$$

### Comparar listas

Supongamos este ejemplo:

```
>>> [1, 2, 3] < [1, 2, 4]
True
```

Python llega a la conclusión de que la lista `[1, 2, 3]` es menor que `[1, 2, 4]` porque va comparando elemento a elemento:

- El `1` es igual en ambas listas.
- El `2` es igual en ambas litas.
- El `3` es menor que el `4`, por lo que la primera lista es menor que la segunda.

Entender la forma en la que se comparan dos listas es importante para poder aplicar otras funciones y obtener los resultados deseados.

**Ver también:**

Esta comparación funciona de forma totalmente análoga a la *comparación de cadenas de texto*.

## 5.1.4 Cuidado con las copias

**Nivel intermedio**

Las listas son estructuras de datos *mutables* y esta característica nos obliga a tener cuidado cuando realizamos copias de listas, ya que la modificación de una de ellas puede afectar a la otra.

Veamos un ejemplo sencillo:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[15, 3, 7, 1]
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/pfi5PC5

**Nota:** A través de *Python Tutor* se puede ver claramente el motivo de por qué ocurre esto. Dado que las variables «apuntan» a la misma zona de memoria, al modificar una de ellas, el cambio también se ve reflejado en la otra.

Una **posible solución** a este problema es hacer una «copia dura». Para ello Python proporciona la función `copy()`:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list.copy()

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[4, 3, 7, 1]
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/Dfi6oLk

Existe **otra aproximación** a este problema, y es utilizar un *troceado* completo de la lista, lo que nos devuelve una «copia desvinculada» de manera implícita:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list[:]

>>> id(original_list) != id(copy_list)
True
```

**Truco:** En el caso de que estemos trabajando con listas que contienen elementos mutables, debemos hacer uso de la función `deepcopy()` dentro del módulo `copy` de la librería estándar.

### 5.1.5 Veracidad múltiple

Si bien podemos usar *sentencias condicionales* para comprobar la veracidad de determinadas expresiones, Python nos ofrece dos funciones «built-in» con las que podemos evaluar si se cumplen **todas** las condiciones `all()` o si se cumple **alguna** condición `any()`. Estas funciones trabajan sobre iterables, y el caso más evidente es una **lista**.

Supongamos un ejemplo en el que queremos comprobar si una determinada palabra cumple las siguientes condiciones:

- Su longitud total es mayor que 4.
- Empieza por «p».
- Contiene, al menos, una «y».

Veamos la **versión clásica**:

```
>>> word = 'python'

>>> if len(word) > 4 and word.startswith('p') and word.count('y') >= 1:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Veamos la **versión con veracidad múltiple** usando `all()`, donde se comprueba que se cumplan **todas** las expresiones:

```
>>> word = 'python'

>>> enough_length = len(word) > 4            # True
>>> right_beginning = word.startswith('p')   # True
>>> min_ys = word.count('y') >= 1            # True

>>> is_cool_word = all([enough_length, right_beginning, min_ys])

>>> if is_cool_word:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Veamos la **versión con veracidad múltiple** usando `any()`, donde se comprueba que se cumpla **alguna** expresión:

```
>>> word = 'yeah'

>>> enough_length = len(word) > 4             # False
>>> right_beginning = word.startswith('p')    # False
>>> min_ys = word.count('y') >= 1             # True

>>> is_fine_word = any([enough_length, right_beginning, min_ys])

>>> if is_fine_word:
...     print('Fine word!')
... else:
...     print('No thanks')
...
Fine word!
```

**Consejo:** Este enfoque puede ser interesante cuando se manejan muchas condiciones o bien cuando queremos separar las condiciones y agruparlas en una única lista.

### 5.1.6 Listas por comprensión

**Nivel intermedio**

Las **listas por comprensión** establecen una técnica para crear listas de forma más **compacta** basándose en el concepto matemático de conjuntos definidos por comprensión.

Podríamos decir que su sintaxis sigue un modelo **VLC (Value-Loop-Condition)** tal y como se muestra en la siguiente figura:

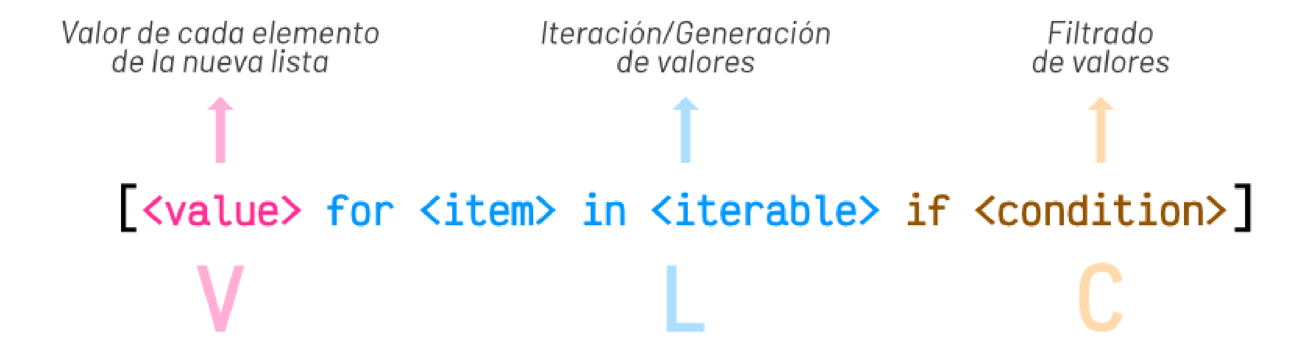



Figura 3: Estructura de una lista por comprensión

En primer lugar veamos un ejemplo en el que convertimos una cadena de texto con valores numéricos en una lista con los mismos valores pero convertidos a enteros. En su **versión clásica** haríamos algo tal que así:

```
>>> values = '32,45,11,87,20,48'

>>> int_values = []

>>> for value in values.split(','):
...     int_value = int(value)
...     int_values.append(int_value)
...

>>> int_values
[32, 45, 11, 87, 20, 48]
```

Ahora veamos el código utilizando una **lista por comprensión**:

```
>>> values = '32,45,11,87,20,48'

>>> int_values = [int(value) for value in values.split(',')]

>>> int_values
[32, 45, 11, 87, 20, 48]
```

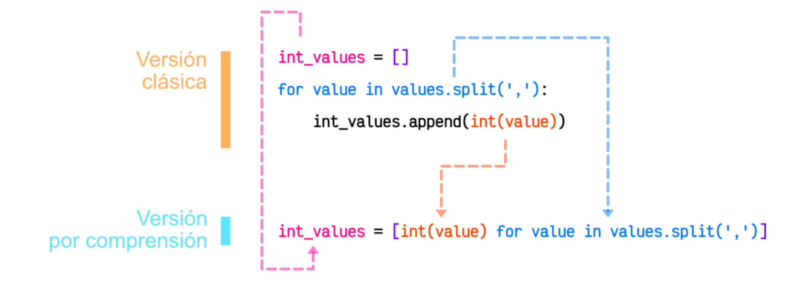



Figura 4: Transformación de estructura clásica en lista por comprensión

## Condiciones en comprensiones

También existe la posibilidad de incluir condiciones en las **listas por comprensión**.

Continuando con el ejemplo anterior, supongamos que sólo queremos crear la lista con aquellos valores que empiecen por el dígito 4:

```
>>> values = '32,45,11,87,20,48'

>>> int_values = [int(v) for v in values.split(',') if v.startswith('4')]

>>> int_values
[45, 48]
```

## Anidamiento en comprensiones

**Nivel avanzado**

En la iteración que usamos dentro de la lista por comprensión es posible usar *bucles anidados*.

Veamos un ejemplo en el que generamos todas las combinaciones de una serie de valores:

```
>>> values = '32,45,11,87,20,48'
>>> svalues = values.split(',')

>>> combinations = [f'{v1}x{v2}' for v1 in svalues for v2 in svalues]

>>> combinations
['32x32',
 '32x45',
 '32x11',
 '32x87',
 '32x20',
 '32x48',
 '45x32',
 '45x45',
 ...
 '48x45',
 '48x11',
 '48x87',
 '48x20',
 '48x48']
```

**Consejo:** Las listas por comprensión son una herramienta muy potente y nos ayuda en muchas ocasiones, pero hay que tener cuidado de no generar **expresiones excesivamente complejas**. En estos casos es mejor una *aproximación clásica*.

**Ejercicio**

Utilizando listas por comprensión, cree una lista que contenga el resultado de aplicar la función $f(x) = 3x + 2$ para $x \in [0, 20)$.

**Salida esperada**: `[2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59]`

### 5.1.7 sys.argv

Cuando queramos ejecutar un programa Python desde **línea de comandos**, tendremos la posibilidad de acceder a los argumentos de dicho programa. Para ello se utiliza una lista que la encontramos dentro del módulo `sys` y que se denomina `argv`:

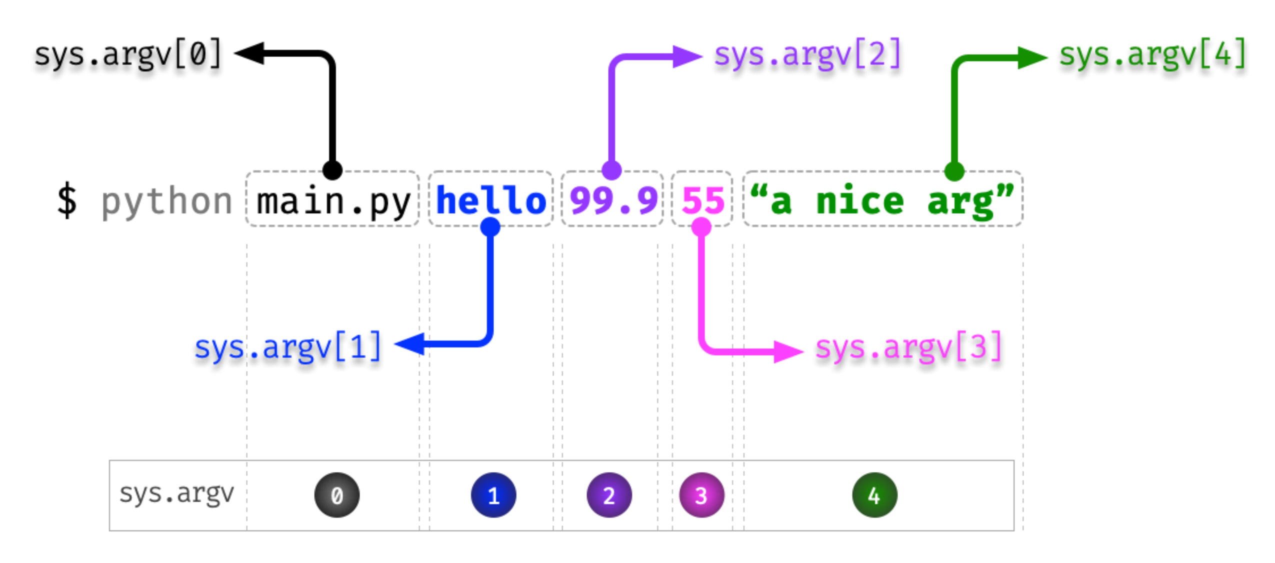



Figura 5: Acceso a parámetros en línea de comandos

Veamos una aplicación de lo anterior en un programa que convierte un número decimal a una determinada base, ambos argumentos pasados por línea de comandos:

dec2base.py

```
import sys

number = int(sys.argv[1])
tobase = int(sys.argv[2])

match tobase:
    case 2:
        result = f'{number:b}'
    case 8:
        result = f'{number:o}'
    case 16:
        result = f'{number:x}'
    case _:
        result = None

if result is None:
    print(f'Base {tobase} not implemented!')
else:
    print(result)
```

Si lo ejecutamos obtenemos lo siguiente:

```
$ python dec2base.py 65535 2
1111111111111111
```

### 5.1.8 Funciones matemáticas

Python nos ofrece, entre otras[4], estas tres funciones matemáticas básicas que se pueden aplicar sobre listas.

**Suma de todos los valores:** Mediante la función `sum()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> sum(data)
28
```

**Mínimo de todos los valores:** Mediante la función `min()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> min(data)
1
```

**Máximo de todos los valores:** Mediante la función `max()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> max(data)
9
```

---

**Ejercicio**

Lea *desde línea de comandos* una serie de números y obtenga la media de dichos valores (*muestre el resultado con 2 decimales*).

La llamada se haría de la siguiente manera:

```
$ python avg.py 32 56 21 99 12 17
```

Plantilla de código para el programa:

```
import sys

# En values tendremos una lista con los valores (como strings)
values = sys.argv[1:]
```

(continué en la próxima página)

[4] Existen multitud de paquetes científicos en Python para trabajar con listas o vectores numéricos. Una de las más famosas es la librería Numpy.

(proviene de la página anterior)

```
# Su código debajo de aquí
```

**Ejemplo**

- Entrada: `32 56 21 99 12 17`
- Salida: `39.50`

### 5.1.9 Listas de listas

**Nivel intermedio**

Como ya hemos visto en varias ocasiones, las listas son estructuras de datos que pueden contener elementos heterogéneos. Estos elementos pueden ser a su vez listas.

A continuación planteamos un ejemplo del mundo deportivo. Un equipo de fútbol suele tener una disposición en el campo organizada en líneas de jugadores. En aquella alineación con la que España ganó la copa del mundo en 2010 había una disposición *4-3-3* con los siguientes jugadores:




Figura 6: Lista de listas (como equipo de fútbol)

Veamos una posible representación de este equipo de fútbol usando una lista compuesta de listas. Primero definimos cada una de las líneas:

```
>>> goalkeeper = 'Casillas'
>>> defenders = ['Capdevila', 'Piqué', 'Puyol', 'Ramos']
>>> midfielders = ['Xavi', 'Busquets', 'X. Alonso']
>>> forwards = ['Iniesta', 'Villa', 'Pedro']
```

Y ahora las juntamos en una única lista:

```
>>> team = [goalkeeper, defenders, midfielders, forwards]

>>> team
['Casillas',
 ['Capdevila', 'Piqué', 'Puyol', 'Ramos'],
 ['Xavi', 'Busquets', 'X. Alonso'],
 ['Iniesta', 'Villa', 'Pedro']]
```

Podemos comprobar el acceso a distintos elementos:

```
>>> team[0]  # portero
'Casillas'

>>> team[1][0]  # lateral izquierdo
'Capdevila'

>>> team[2]  # centrocampistas
['Xavi', 'Busquets', 'X. Alonso']

>>> team[3][1]  # delantero centro
'Villa'
```

---

**Ejercicio**

Escriba un programa que permita multiplicar únicamente matrices de 2 filas por 2 columnas. Veamos un ejemplo concreto:

```
A = [[6, 4], [8, 9]]
B = [[3, 2], [1, 7]]
```

El producto $\mathbb{P} = A \times B$ se calcula siguiendo la multiplicación de matrices tal y como se indica a continuación:

$$\mathbb{P} = \begin{pmatrix} 6_{[00]} & 4_{[01]} \\ 8_{[10]} & 9_{[11]} \end{pmatrix} \times \begin{pmatrix} 3_{[00]} & 2_{[01]} \\ 1_{[10]} & 7_{[11]} \end{pmatrix} =$$

$$\begin{pmatrix} 6 \cdot 3 + 4 \cdot 1 & 6 \cdot 2 + 4 \cdot 7 \\ 8 \cdot 3 + 9 \cdot 1 & 8 \cdot 2 + 9 \cdot 7 \end{pmatrix} = \begin{pmatrix} 22 & 40 \\ 33 & 79 \end{pmatrix}$$

## EJERCICIOS DE REPASO

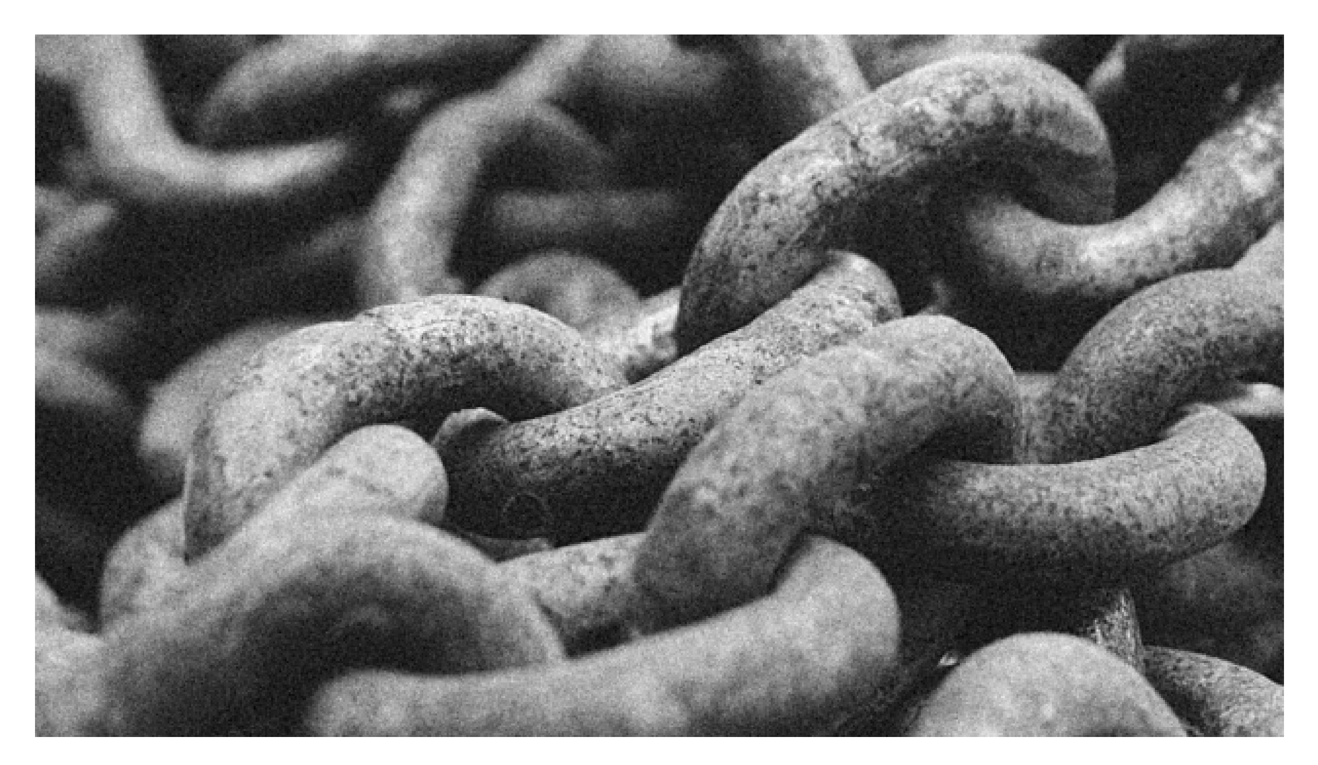
1. pycheck: **max_value**
2. pycheck: **remove_dups**
3. pycheck: **flatten_list**
4. pycheck: **remove_consecutive_dups**
5. pycheck: **all_same**
6. pycheck: **sum_diagonal**
7. pycheck: **powers2**
8. pycheck: **dec2bin**
9. pycheck: **sum_mixed**
10. pycheck: **n_mult**
11. pycheck: **remove_second**
12. pycheck: **nth_power**
13. pycheck: **name_initials**
14. pycheck: **non_consecutive**
15. pycheck: **mult_reduce**
16. pycheck: **digit_rev_list**
17. pycheck: **time_plus_minutes**
18. pycheck: **sum_positive**
19. pycheck: **add_inverse**
20. pycheck: **descending_numbers**
21. pycheck: **merge_sorted**
22. pycheck: **min_value**
23. pycheck: **trimmed_sum**
24. pycheck: **wolves**
25. pycheck: **maxmin_values**
26. pycheck: **cascading_subsets**

27. pycheck: **diff_cuboid**
28. pycheck: **fl_strip**
29. pycheck: **logical_chain**
30. pycheck: **smallest_unused_id**
31. pycheck: **find_odds**
32. pycheck: **reagent_formula**
33. pycheck: **whats_next**
34. pycheck: **npartition**
35. pycheck: **add_length**
36. pycheck: **reversing_words**
37. pycheck: **barycenter**

## AMPLIAR CONOCIMIENTOS

- Linked Lists in Python: An Introduction
- Python Command Line Arguments
- Sorting Data With Python
- When to Use a List Comprehension in Python
- Using the Python zip() Function for Parallel Iteration
- Lists and Tuples in Python
- How to Use sorted() and sort() in Python
- Using List Comprehensions Effectively

## 5.2 Tuplas



El concepto de **tupla** es muy similar al de *lista*. Aunque hay algunas diferencias menores, lo fundamental es que, mientras una *lista* es mutable y se puede modificar, una *tupla* no admite cambios y por lo tanto, es **inmutable**.[1]

### 5.2.1 Creando tuplas

Podemos pensar en crear tuplas tal y como *lo hacíamos con listas*, pero usando **paréntesis** en lugar de *corchetes*:

```
>>> empty_tuple = ()

>>> tenerife_geoloc = (28.46824, -16.25462)

>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
```

**Truco:** Al igual que con las listas, las tuplas admiten diferentes tipos de datos: `('a', 1, True)`

[1] Foto original de portada por engin akyurt en Unsplash.

### Tuplas de un elemento

Hay que prestar especial atención cuando vamos a crear una **tupla de un único elemento**. La intención primera sería hacerlo de la siguiente manera:

```
>>> one_item_tuple = ('Papá Noel')

>>> one_item_tuple
'Papá Noel'

>>> type(one_item_tuple)
str
```

Realmente, hemos creado una variable de tipo `str` (cadena de texto). Para crear una tupla de un elemento debemos añadir una **coma** al final:

```
>>> one_item_tuple = ('Papá Noel',)

>>> one_item_tuple
('Papá Noel',)

>>> type(one_item_tuple)
tuple
```

### Tuplas sin paréntesis

Según el caso, hay veces que nos podemos encontrar con tuplas que no llevan paréntesis. Quizás no está tan extendido, pero a efectos prácticos tiene el mismo resultado. Veamos algunos ejemplos de ello:

```
>>> one_item_tuple = 'Papá Noel',

>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'

>>> tenerife_geoloc = 28.46824, -16.25462
```

**Advertencia:** Aunque está permitido, **NUNCA** llames `tuple` a una variable porque destruirías la función que nos permite crear tuplas. Y tampoco uses nombres derivados como `_tuple` o `tuple_` ya que no son nombres representativos que *identifiquen el propósito de la variable*.

### 5.2.2 Modificar una tupla

Como ya hemos comentado previamente, las tuplas con estructuras de datos **inmutables**. Una vez que las creamos con un valor, no podemos modificarlas. Veamos qué ocurre si lo intentamos:

```
>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'

>>> three_wise_men[0] = 'Tom Hanks'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

### 5.2.3 Conversión

Para convertir otros tipos de datos en una tupla podemos usar la función `tuple()`:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']

>>> tuple(shopping)
('Agua', 'Aceite', 'Arroz')
```

Esta conversión es válida para aquellos tipos de datos que sean *iterables*: cadenas de caracteres, listas, diccionarios, conjuntos, etc. Un ejemplo que no funciona es intentar convertir un número en una tupla:

```
>>> tuple(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

El uso de la función `tuple()` sin argumentos equivale a crear una tupla vacía:

```
>>> tuple()
()
```

**Truco:** Para crear una tupla vacía, se suele recomendar el uso de `()` frente a `tuple()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

### 5.2.4 Operaciones con tuplas

Con las tuplas podemos realizar *todas las operaciones que vimos con listas* **salvo las que conlleven una modificación** «in-situ» de la misma:

- `reverse()`
- `append()`
- `extend()`
- `remove()`
- `clear()`
- `sort()`

---

**Truco:** Sí es posible aplicar `sorted()` o `reversed()` sobre una tupla ya que no estamos modificando su valor sino creando un nuevo objeto.

---

**Ver también:**

La comparación de tuplas funciona exactamente igual que la *comparación de listas*.

### 5.2.5 Desempaquetado de tuplas

El **desempaquetado** es una característica de las tuplas que nos permite *asignar una tupla a variables independientes*:

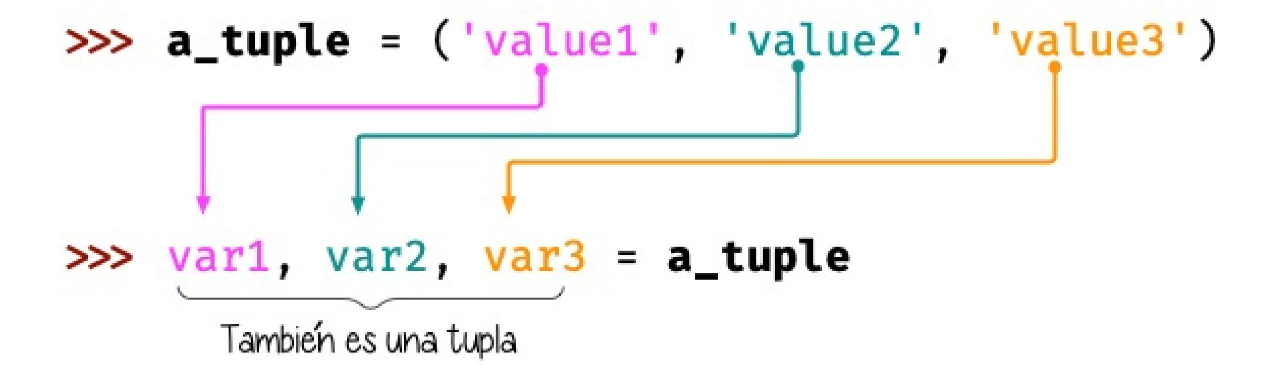



Figura 7: Desempaquetado de tuplas

Veamos un ejemplo con código:

```
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')

>>> king1, king2, king3 = three_wise_men

>>> king1
'Melchor'
>>> king2
'Gaspar'
>>> king3
'Baltasar'
```

Python proporciona la función «built-in» `divmod()` que devuelve el cociente y el resto de una división usando una única llamada. Lo interesante (para el caso que nos ocupa) es que se suele utilizar el desempaquetado de tuplas para obtener los valores:

```
>>> quotient, remainder = divmod(7, 3)

>>> quotient
2
>>> remainder
1
```

### Intercambio de valores

A través del desempaquetado de tuplas podemos llevar a cabo *el intercambio de los valores de dos variables* de manera directa:

```
>>> value1 = 40
>>> value2 = 20

>>> value1, value2 = value2, value1

>>> value1
20
>>> value2
40
```

**Nota:** A priori puede parecer que esto es algo «natural», pero en la gran mayoría de lenguajes de programación no es posible hacer este intercambio de forma «directa» ya que necesitamos recurrir a una tercera variable «auxiliar» como almacén temporal en el paso intermedio de traspaso de valores.

### Desempaquetado extendido

No tenemos que ceñirnos a realizar desempaquetado uno a uno. También podemos extenderlo e indicar ciertos «grupos» de elementos mediante el operador *.

Veamos un ejemplo:

```
>>> ranking = ('G', 'A', 'R', 'Y', 'W')

>>> head, *body, tail = ranking

>>> head
'G'

>>> body
['A', 'R', 'Y']

>>> tail
'W'
```

### Desempaquetado genérico

El desempaquetado de tuplas es extensible a cualquier tipo de datos que sea **iterable**. Veamos algunos ejemplos de ello.

Sobre cadenas de texto:

```
>>> oxygen = 'O2'
>>> first, last = oxygen
>>> first, last
('O', '2')

>>> text = 'Hello, World!'
>>> head, *body, tail = text
>>> head, body, tail
('H', ['e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd'], '!')
```

Sobre listas:

```
>>> writer1, writer2, writer3 = ['Virginia Woolf', 'Jane Austen', 'Mary Shelley']
>>> writer1, writer2, writer3
('Virginia Woolf', 'Jane Austen', 'Mary Shelley')

>>> text = 'Hello, World!'
>>> word1, word2 = text.split()
>>> word1, word2
('Hello,', 'World!')
```

### 5.2.6 ¿Tuplas por comprensión?

Los tipos de datos mutables (*listas, diccionarios y conjuntos*) sí permiten comprensiones pero no así los tipos de datos inmutables como *cadenas de texto* y *tuplas*.

Si intentamos crear una **tupla por comprensión** utilizando paréntesis alrededor de la expresión, vemos que no obtenemos ningún error al ejecutarlo:

```
>>> myrange = (number for number in range(1, 6))
```

Sin embargo no hemos conseguido una tupla por comprensión sino un generador:

```
>>> myrange
<generator object <genexpr> at 0x10b3732e0>
```

### 5.2.7 Tuplas vs Listas

Aunque puedan parecer estructuras de datos muy similares, sabemos que las tuplas carecen de ciertas operaciones, especialmente las que tienen que ver con la modificación de sus valores, ya que no son inmutables. Si las listas son más flexibles y potentes, ¿por qué íbamos a necesitar tuplas? Veamos 4 potenciales ventajas del uso de tuplas frente a las listas:

1. Las tuplas ocupan **menos espacio** en memoria.
2. En las tuplas existe **protección** frente a cambios indeseados.
3. Las tuplas se pueden usar como **claves de diccionarios** (son *«hashables»*).
4. Las namedtuples son una alternativa sencilla a los objetos.

## 5.3 Diccionarios



Podemos trasladar el concepto de *diccionario* de la vida real al de *diccionario* en Python. Al fin y al cabo un diccionario es un objeto que contiene palabras, y cada palabra tiene asociado un significado. Haciendo el paralelismo, diríamos que en Python un diccionario es también un objeto indexado por **claves** (las palabras) que tienen asociados unos **valores** (los significados).[1]

Los diccionarios en Python tienen las siguientes *características*:

- Mantienen el **orden** en el que se insertan las claves.[2]
- Son **mutables**, con lo que admiten añadir, borrar y modificar sus elementos.
- Las **claves** deben ser **únicas**. A menudo se utilizan las *cadenas de texto* como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- Tienen un **acceso muy rápido** a sus elementos, debido a la forma en la que están implementados internamente.[3]

**Nota:** En otros lenguajes de programación, a los diccionarios se les conoce como *arrays*

[1] Foto original de portada por Aaron Burden en Unsplash.

[2] Aunque históricamente Python no establecía que las claves de los diccionarios tuvieran que mantener su orden de inserción, a partir de Python 3.7 este comportamiento cambió y se garantizó el orden de inserción de las claves como parte oficial de la especificación del lenguaje.

[3] Véase este análisis de complejidad y rendimiento de distintas estructuras de datos en CPython.

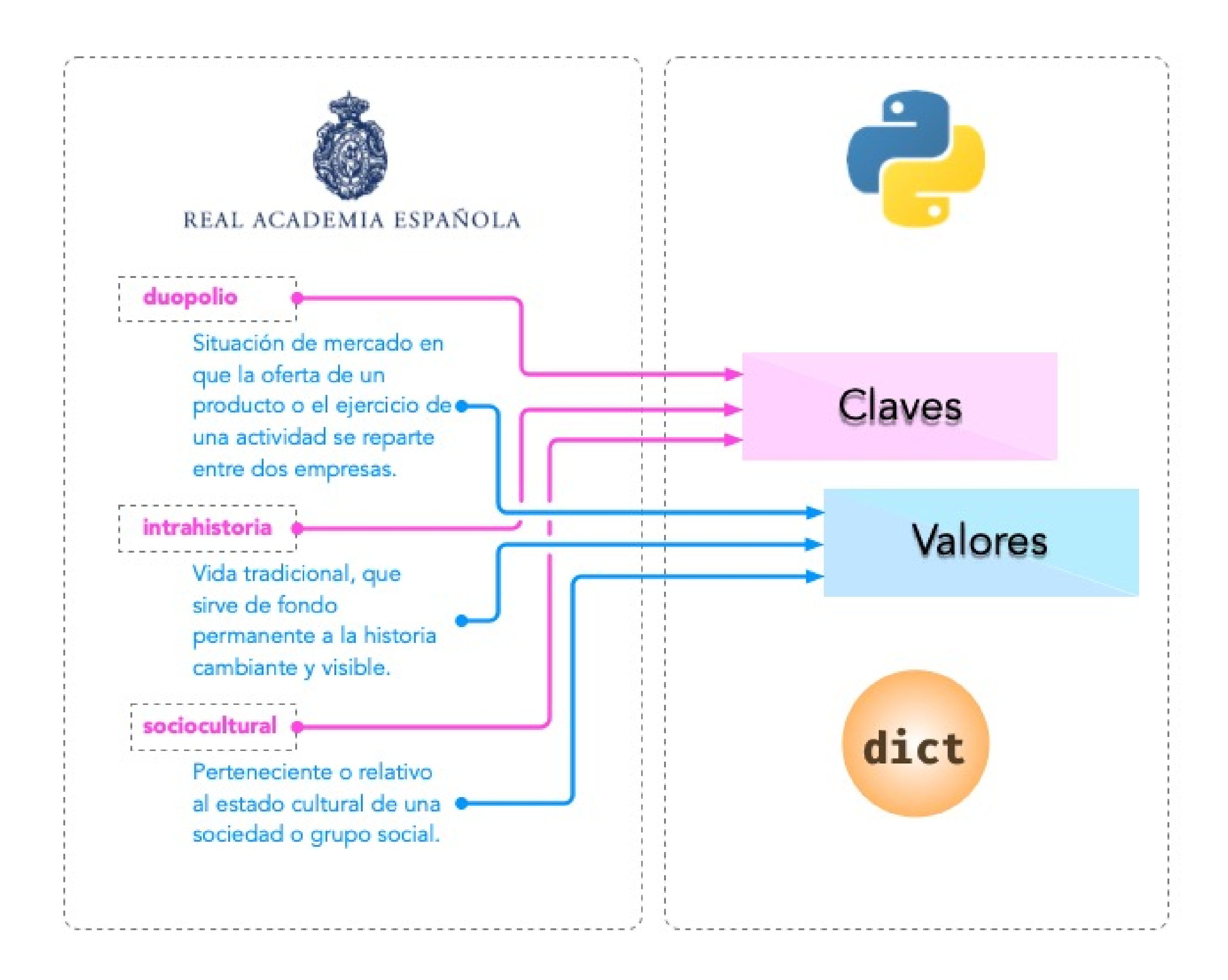



Figura 8: Analogía de un diccionario en Python

*asociativos*, *«hashes»* o *«hashmaps»*.

### 5.3.1 Creando diccionarios

Para crear un diccionario usamos llaves `{}` rodeando asignaciones `clave: valor` que están separadas por comas. Veamos algunos ejemplos de diccionarios:

```
>>> empty_dict = {}

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> population_can = {
...     2015: 2_135_209,
...     2016: 2_154_924,
...     2017: 2_177_048,
...     2018: 2_206_901,
...     2019: 2_220_270
... }
```

En el código anterior podemos observar la creación de un diccionario vacío, otro donde sus claves y sus valores son cadenas de texto y otro donde las claves y los valores son valores enteros.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/Sfav2Yw

**Advertencia:** Aunque está permitido, **NUNCA** llames `dict` a una variable porque destruirías la función que nos permite crear diccionarios. Y tampoco uses nombres derivados como `_dict` o `dict_` ya que no son nombres representativos que *identifiquen el propósito de la variable*.

**Ejercicio**

Cree un diccionario con los nombres de 5 personas de su familia y sus edades.

### 5.3.2 Conversión

Para convertir otros tipos de datos en un diccionario podemos usar la función `dict()`:

```
>>> # Diccionario a partir de una lista de cadenas de texto
>>> dict(['a1', 'b2'])
{'a': '1', 'b': '2'}

>>> # Diccionario a partir de una tupla de cadenas de texto
>>> dict(('a1', 'b2'))
{'a': '1', 'b': '2'}

>>> # Diccionario a partir de una lista de listas
>>> dict([['a', 1], ['b', 2]])
{'a': 1, 'b': 2}
```

**Nota:** Si nos fijamos bien, cualquier iterable que tenga una estructura interna de 2 elementos es susceptible de convertirse en un diccionario a través de la función `dict()`.

#### Diccionario vacío

Existe una manera particular de usar `dict()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en un diccionario, con lo que obtendremos un *diccionario vacío*:

```
>>> dict()
{}
```

**Truco:** Para crear un diccionario vacío, se suele recomendar el uso de `{}` frente a `dict()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

#### Creación con `dict()`

También es posible utilizar la función `dict()` para crear dicionarios y no tener que utilizar llaves y comillas:

Supongamos que queremos transformar la siguiente tabla en un diccionario:

| Atributo | Valor |
|---|---|
| name | Guido |
| surname | Van Rossum |
| job | Python creator |

Utilizando la construcción mediante `dict` podemos pasar clave y valor como **argumentos** de la función:

```
>>> person = dict(
...     name='Guido',
...     surname='Van Rossum',
...     job='Python creator'
... )

>>> person
{'name': 'Guido', 'surname': 'Van Rossum', 'job': 'Python creator'}
```

El inconveniente que tiene esta aproximación es que las **claves deben ser identificadores válidos** en Python. Por ejemplo, no se permiten espacios:

```
>>> person = dict(
...     name='Guido van Rossum',
...     date of birth='31/01/1956'
  File "<stdin>", line 3
    date of birth='31/01/1956'
         ^
SyntaxError: invalid syntax
```

### Nivel intermedio

Es posible crear un diccionario especificando sus claves y un único valor de «relleno»:

```
>>> dict.fromkeys('aeiou', 0)
{'a': 0, 'e': 0, 'i': 0, 'o': 0, 'u': 0}
```

---

**Nota:** Es válido pasar cualquier «iterable» como referencia a las claves.

---

### 5.3.3 Operaciones con diccionarios

#### Obtener un elemento

Para obtener un elemento de un diccionario basta con escribir la **clave** entre corchetes. Veamos un ejemplo:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae['anarcoide']
'Que tiende al desorden'
```

Si intentamos acceder a una clave que no existe, obtendremos un error:

```
>>> rae['acceso']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'acceso'
```

#### Usando `get()`

Existe una función muy útil para «superar» los posibles errores de acceso por claves inexistentes. Se trata de `get()` y su comportamiento es el siguiente:

1. Si la clave que buscamos existe, nos devuelve su valor.
2. Si la clave que buscamos no existe, nos devuelve `None`[4] salvo que le indiquemos otro valor por defecto, pero en ninguno de los dos casos obtendremos un error.

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

>>> rae.get('bifronte')
'De dos frentes o dos caras'

>>> rae.get('programación')

```

(continué en la próxima página)

[4] `None` es la palabra reservada en Python para la «nada». Más información en esta web.

(proviene de la página anterior)

```
11 >>> rae.get('programación', 'No disponible')
12 'No disponible'
```

**Línea 6:** Equivalente a `rae['bifronte']`.

**Línea 9:** La clave buscada no existe y obtenemos `None`.[5]

**Línea 11:** La clave buscada no existe y nos devuelve el valor que hemos aportado por defecto.

### Añadir o modificar un elemento

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la *clave* y asignarle un *valor*:

- Si la clave **ya existía** en el diccionario, **se reemplaza** el valor existente por el nuevo.
- Si la clave **es nueva**, **se añade** al diccionario con su valor. *No vamos a obtener un error a diferencia de las listas.*

Partimos del siguiente diccionario para ejemplificar estas acciones:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }
```

Vamos a **añadir** la palabra *enjuiciar* a nuestro diccionario de la Real Academia de La Lengua:

```
>>> rae['enjuiciar'] = 'Someter una cuestión a examen, discusión y juicio'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'}
```

Supongamos ahora que queremos **modificar** el significado de la palabra *enjuiciar* por otra acepción:

[5] Realmente no estamos viendo nada en la consola de Python porque la representación en cadena de texto es vacía.

```
>>> rae['enjuiciar'] = 'Instruir, juzgar o sentenciar una causa'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

### Creando desde vacío

Una forma muy habitual de trabajar con diccionarios es utilizar el **patrón creación** partiendo de uno vacío e ir añadiendo elementos poco a poco.

Supongamos un ejemplo en el que queremos construir un diccionario donde las claves son las letras vocales y los valores son sus posiciones:

```
>>> VOWELS = 'aeiou'

>>> enum_vowels = {}

>>> for i, vowel in enumerate(VOWELS, start=1):
...     enum_vowels[vowel] = i
...

>>> enum_vowels
{'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}
```

**Nota:** Hemos utilizando la función `enumerate()` que ya vimos para las listas en el apartado: *Iterar usando enumeración*.

**Ejercicio**

pycheck: **cities**

### Pertenencia de una clave

La forma **pitónica** de comprobar la existencia de una clave dentro de un diccionario, es utilizar el operador `in`:

```
>>> 'bifronte' in rae
True

>>> 'almohada' in rae
False

>>> 'montuvio' not in rae
False
```

**Nota:** El operador `in` siempre devuelve un valor booleano, es decir, verdadero o falso.

**Ejercicio**

pycheck: **count_letters**

### Obtener todos los elementos

Python ofrece mecanismos para obtener todos los elementos de un diccionario. Partimos del siguiente diccionario:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

**Obtener todas las claves de un diccionario:** Mediante la función `keys()`:

```
>>> rae.keys()
dict_keys(['bifronte', 'anarcoide', 'montuvio', 'enjuiciar'])
```

**Obtener todos los valores de un diccionario:** Mediante la función `values()`:

```
>>> rae.values()
dict_values([
    'De dos frentes o dos caras',
    'Que tiende al desorden',
```

(continué en la próxima página)

(proviene de la página anterior)

```
    'Campesino de la costa',
    'Instruir, juzgar o sentenciar una causa'
])
```

**Obtener todos los pares «clave-valor» de un diccionario:** Mediante la función `items()`:

```
>>> rae.items()
dict_items([
    ('bifronte', 'De dos frentes o dos caras'),
    ('anarcoide', 'Que tiende al desorden'),
    ('montuvio', 'Campesino de la costa'),
    ('enjuiciar', 'Instruir, juzgar o sentenciar una causa')
])
```

**Nota:** Para este último caso cabe destacar que los «items» se devuelven como una lista de *tuplas*, donde cada tupla tiene dos elementos: el primero representa la clave y el segundo representa el valor.

### Longitud de un diccionario

Podemos conocer el número de elementos («clave-valor») que tiene un diccionario con la función `len()`:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

>>> len(rae)
4
```

### Iterar sobre un diccionario

En base a *los elementos que podemos obtener*, Python nos proporciona tres maneras de iterar sobre un diccionario.

**Iterar sobre claves:**

```
>>> for word in rae.keys():
...     print(word)
...
bifronte
anarcoide
montuvio
enjuiciar
```

**Iterar sobre valores**:

```
>>> for meaning in rae.values():
...     print(meaning)
...
De dos frentes o dos caras
Que tiende al desorden
Campesino de la costa
Instruir, juzgar o sentenciar una causa
```

**Iterar sobre «clave-valor»**:

```
>>> for word, meaning in rae.items():
...     print(f'{word}: {meaning}')
...
bifronte: De dos frentes o dos caras
anarcoide: Que tiende al desorden
montuvio: Campesino de la costa
enjuiciar: Instruir, juzgar o sentenciar una causa
```

**Nota:** En este último caso, recuerde el uso de los *«f-strings»* para formatear cadenas de texto.

**Ejercicio**

pycheck: **avg_population**

### Combinar diccionarios

Dados dos (o más) diccionarios, es posible «mezclarlos» para obtener una combinación de los mismos. Esta combinación se basa en dos premisas:

1. Si la clave no existe, se añade con su valor.
2. Si la clave ya existe, se añade con el valor del «último» diccionario en la mezcla.[6]

Python ofrece dos mecanismos para realizar esta combinación. Vamos a partir de los siguientes diccionarios para ejemplificar su uso:

```
>>> rae1 = {
...     'bifronte': 'De dos frentes o dos caras',
...     'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'
... }

>>> rae2 = {
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa',
...     'enjuiciar': 'Instruir, juzgar o sentenciar una causa'
... }
```

**Sin modificar los diccionarios originales:** Mediante el operador `**`:

```
>>> {**rae1, **rae2}
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

A partir de **Python 3.9** podemos utilizar el operador `|` para combinar dos diccionarios:

```
>>> rae1 | rae2
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

**Modificando los diccionarios originales:** Mediante la función `update()`:

```
>>> rae1.update(rae2)

>>> rae1
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
```

(continué en la próxima página)

[6] En este caso «último» hace referencia al diccionario que se encuentra más a la derecha en la expresión.

(proviene de la página anterior)

```
'anarcoide': 'Que tiende al desorden',
'montuvio': 'Campesino de la costa'}
```

**Nota:** Tener en cuenta que el orden en el que especificamos los diccionarios a la hora de su combinación (mezcla) es relevante en el resultado final. En este caso *el orden de los factores sí altera el producto*.

### Borrar elementos

Python nos ofrece, al menos, tres formas para borrar elementos en un diccionario:

**Por su clave:** Mediante la sentencia `del`:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> del rae['bifronte']

>>> rae
{'anarcoide': 'Que tiende al desorden', 'montuvio': 'Campesino de la costa'}
```

**Por su clave (con extracción):** Mediante la función `pop()` podemos extraer un elemento del diccionario por su clave. Vendría a ser una combinación de `get()` + `del`:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae.pop('anarcoide')
'Que tiende al desorden'

>>> rae
{'bifronte': 'De dos frentes o dos caras', 'montuvio': 'Campesino de la costa'}

>>> rae.pop('bucle')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'bucle'
```

**Advertencia:** Si la clave que pretendemos extraer con `pop()` no existe, obtendremos un error.

**Borrado completo del diccionario:**

1. Utilizando la función `clear()`:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae.clear()

>>> rae
{}
```

2. «Reinicializando» el diccionario a vacío con `{}`:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae = {}

>>> rae
{}
```

**Nota:** La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

### 5.3.4 Cuidado con las copias

#### Nivel intermedio

Al igual que ocurría con *las listas*, si hacemos un cambio en un diccionario, se verá reflejado en todas las variables que hagan referencia al mismo. Esto se deriva de su propiedad de ser *mutable*. Veamos un ejemplo concreto:

```
>>> original_rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> copy_rae = original_rae

>>> original_rae['bifronte'] = 'bla bla bla'

>>> original_rae
{'bifronte': 'bla bla bla',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

>>> copy_rae
{'bifronte': 'bla bla bla',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

Una **posible solución** a este problema es hacer una «copia dura». Para ello Python proporciona la función `copy()`:

```
>>> original_rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> copy_rae = original_rae.copy()

>>> original_rae['bifronte'] = 'bla bla bla'

>>> original_rae
{'bifronte': 'bla bla bla',
'anarcoide': 'Que tiende al desorden',
'montuvio': 'Campesino de la costa'}

>>> copy_rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

**Truco:** En el caso de que estemos trabajando con diccionarios que contienen elementos mutables, debemos hacer uso de la función `deepcopy()` dentro del módulo `copy` de la librería

estándar.

### 5.3.5 Diccionarios por comprensión

**Nivel intermedio**

De forma análoga a cómo se escriben las *listas por comprensión*, podemos aplicar este método a los diccionarios usando llaves `{ }`.

Veamos un ejemplo en el que creamos un **diccionario por comprensión** donde las claves son palabras y los valores son sus longitudes:

```
>>> words = ('sun', 'space', 'rocket', 'earth')

>>> words_length = {word: len(word) for word in words}

>>> words_length
{'sun': 3, 'space': 5, 'rocket': 6, 'earth': 5}
```

También podemos aplicar **condiciones** a estas comprensiones. Continuando con el ejemplo anterior, podemos incorporar la restricción de sólo incluir palabras que no empiecen por vocal:

```
>>> words = ('sun', 'space', 'rocket', 'earth')

>>> words_length = {w: len(w) for w in words if w[0] not in 'aeiou'}

>>> words_length
{'sun': 3, 'space': 5, 'rocket': 6}
```

**Nota:** Se puede consultar el PEP-274 para ver más ejemplos sobre diccionarios por comprensión.

**Ejercicio**

pycheck: **split_marks**

### 5.3.6 Objetos «hashables»

**Nivel avanzado**

La única restricción que deben cumplir las **claves** de un diccionario es ser **«hashables»**[7]. Un objeto es «hashable» si se le puede asignar un valor «hash» que no cambia en ejecución durante toda su vida.

Para encontrar el «hash» de un objeto, Python usa la función `hash()`, que devuelve un número entero y es utilizado para indexar la *tabla «hash»* que se mantiene internamente:

```
>>> hash(999)
999

>>> hash(3.14)
322818021289917443

>>> hash('hello')
-8103770210014465245

>>> hash(('a', 'b', 'c'))
-2157188727417140402
```

Para que un objeto sea «hashable», debe ser **inmutable**:

```
>>> hash(['a', 'b', 'c'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

---

**Nota:** De lo anterior se deduce que las claves de los diccionarios, al tener que ser «hasheables», sólo pueden ser objetos inmutables.

---

La función «built-in» `hash()` realmente hace una llamada al método mágico `__hash__()` del objeto en cuestión:

```
>>> hash('spiderman')
-8105710090476541603

>>> 'spiderman'.__hash__()
-8105710090476541603
```

---

[7] Se recomienda esta ponencia de Víctor Terrón sobre objetos «hashables».

### EJERCICIOS DE REPASO

1. pycheck: **group_words**
2. pycheck: **same_dict_values**
3. pycheck: **build_super_dict**
4. pycheck: **clear_dict_values**
5. pycheck: **fix_keys**
6. pycheck: **order_stock**
7. pycheck: **inventory_moves**
8. pycheck: **merge_dicts**
9. pycheck: **sort_dict**
10. pycheck: **money_back**
11. pycheck: **money_back_max**

### AMPLIAR CONOCIMIENTOS

- Using the Python defaultdict Type for Handling Missing Keys
- Python Dictionary Iteration: Advanced Tips & Tricks
- Python KeyError Exceptions and How to Handle Them
- Dictionaries in Python
- How to Iterate Through a Dictionary in Python
- Shallow vs Deep Copying of Python Objects

## 5.4 Conjuntos



Un **conjunto** en Python representa una serie de **valores únicos** y **sin orden establecido**, con la única restricción de que sus elementos deben ser *«hashables»*. Mantiene muchas similitudes con el concepto matemático de conjunto[1]

### 5.4.1 Creando conjuntos

Para crear un conjunto basta con separar sus valores por *comas* y rodearlos de llaves {}:

```
>>> lottery = {21, 10, 46, 29, 31, 94}

>>> lottery
{10, 21, 29, 31, 46, 94}
```

La excepción la tenemos a la hora de crear un **conjunto vacío**, ya que, siguiendo la lógica de apartados anteriores, deberíamos hacerlo a través de llaves:

```
>>> wrong_empty_set = {}

>>> type(wrong_empty_set)
dict
```

[1] Foto original de portada por Duy Pham en Unsplash.

**Advertencia:** Si hacemos esto, lo que obtenemos es un *diccionario vacío*.

La única opción que tenemos es utilizar la función `set()`:

```
>>> empty_set = set()

>>> empty_set
set()

>>> type(empty_set)
set
```

**Advertencia:** Aunque está permitido, **NUNCA** llames `set` a una variable porque destruirías la función que nos permite crear conjuntos. Y tampoco uses nombres derivados como `_set` o `set_` ya que no son nombres representativos que *identifiquen el propósito de la variable*.

### 5.4.2 Conversión

Para convertir otros tipos de datos en un conjunto podemos usar la función `set()` sobre cualquier iterable:

```
>>> set('aplatanada')
{'a', 'd', 'l', 'n', 'p', 't'}

>>> set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5])
{1, 2, 3, 4, 5}

>>> set(('ADENINA', 'TIMINA', 'TIMINA', 'GUANINA', 'ADENINA', 'CITOSINA'))
{'ADENINA', 'CITOSINA', 'GUANINA', 'TIMINA'}

>>> set({'manzana': 'rojo', 'plátano': 'amarillo', 'kiwi': 'verde'})
{'kiwi', 'manzana', 'plátano'}
```

**Importante:** Como se ha visto en los ejemplos anteriores, `set()` se suele utilizar en muchas ocasiones como una forma de **extraer los valores únicos** de otros tipos de datos. En el caso de los diccionarios se extraen las claves, que, por definición, son únicas.

**Nota:** El hecho de que en los ejemplos anteriores los elementos de los conjuntos estén

ordenados es únicamente un «detalle de implementación» en el que no se puede confiar.

### 5.4.3 Operaciones con conjuntos

#### Obtener un elemento

En un conjunto no existe un orden establecido para sus elementos, por lo tanto **no podemos acceder a un elemento en concreto**.

De este hecho se deriva igualmente que **no podemos modificar un elemento existente**, ya que ni siquiera tenemos acceso al mismo. Python sí nos permite añadir o borrar elementos de un conjunto.

#### Añadir un elemento

Para añadir un elemento a un conjunto debemos utilizar la función `add()`. Como ya hemos indicado, al no importar el orden dentro del conjunto, la inserción no establece a priori la posición donde se realizará.

A modo de ejemplo, vamos a partir de un conjunto que representa a los cuatro integrantes originales de *The Beatles*. Luego añadiremos a un nuevo componente:

```
>>> # John Lennon, Paul McCartney, George Harrison y Ringo Starr
>>> beatles = set(['Lennon', 'McCartney', 'Harrison', 'Starr'])

>>> beatles.add('Best')  # Pete Best

>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/E8Q3p1m

**Truco:** Una pequeña **regla mnemotécnica** para diferenciar `add()` de `append()` es que la función `append()` significa añadir al final, y como los conjuntos no mantienen un orden, esta función se aplica únicamente a listas. Por descarte, la función `add()` se aplica sobre conjuntos.

**Ejercicio**

pycheck: **tupleset**

### Borrar elementos

Para borrar un elemento de un conjunto podemos utilizar la función `remove()`. Siguiendo con el ejemplo anterior, vamos a borrar al último «beatle» añadido:

```
>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> beatles.remove('Best')

>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

### Longitud de un conjunto

Podemos conocer el número de elementos (*cardinalidad*) que tiene un conjunto con la función `len()`:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> len(beatles)
4
```

### Iterar sobre un conjunto

Tal y como hemos visto para otros tipos de datos *iterables*, la forma de recorrer los elementos de un conjunto es utilizar la sentencia `for`:

```
>>> for beatle in beatles:
...     print(beatle)
...
Harrison
McCartney
Starr
Lennon
```

**Consejo:** Como en el ejemplo anterior, es muy común utilizar una *variable en singular* para recorrer un iterable (en plural). No es una regla fija ni sirve para todos los casos, pero sí suele ser una *buena práctica*.

### Pertenencia de elemento

Al igual que con otros tipos de datos, Python nos ofrece el operador `in` para determinar si un elemento pertenece a un conjunto:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> 'Lennon' in beatles
True

>>> 'Fari' in beatles
False
```

### Ordenando un conjunto

Ya hemos comentado que los conjuntos **no mantienen un orden**. ¿Pero qué ocurre si intentamos ordenarlo?

```
>>> marks = {8, 4, 6, 2, 9, 5}

>>> sorted(marks)
[2, 4, 5, 6, 8, 9]
```

Obtenemos **una lista con los elementos ordenados**.

Hay que tener en cuenta que, lógicamente, no podremos hacer uso de la función `sort()` sobre un conjunto:

```
>>> marks.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'set' object has no attribute 'sort'
```

## 5.4.4 Teoría de conjuntos

Vamos a partir de dos conjuntos $A = \{1, 2\}$ y $B = \{2, 3\}$ para ejemplificar las distintas operaciones que se pueden hacer entre ellos basadas en los Diagramas de Venn y la Teoría de Conjuntos:

```
>>> A = {1, 2}

>>> B = {2, 3}
```

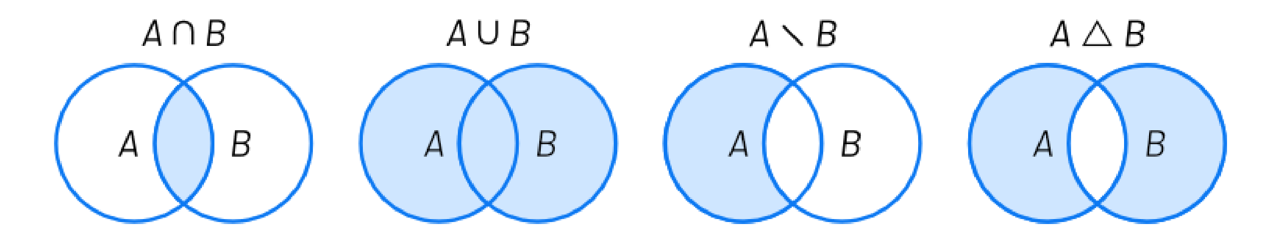



Figura 9: Diagramas de Venn

### Intersección

$A \cap B$ – Elementos que están a la vez en $A$ y en $B$:

```
>>> A & B
{2}

>>> A.intersection(B)
{2}
```

### Unión

$A \cup B$ – Elementos que están tanto en $A$ como en $B$:

```
>>> A | B
{1, 2, 3}

>>> A.union(B)
{1, 2, 3}
```

### Diferencia

$A \setminus B$ – Elementos que están en $A$ y no están en $B$:

```
>>> A - B
{1}

>>> A.difference(B)
{1}
```

### Diferencia simétrica

$A \triangle B$ – Elementos que están en $A$ o en $B$ pero no en ambos conjuntos:

```
>>> A ^ B
{1, 3}

>>> A.symmetric_difference(B)
{1, 3}
```

Podemos comprobar que se cumple la siguiente igualdad matemática $A \triangle B = (A \setminus B) \cup (B \setminus A)$:

```
>>> A ^ B == (A - B) | (B - A)
True
```

### Inclusión

- Un conjunto $B$ es un **subconjunto** de otro conjunto $A$ si todos los elementos de $B$ están incluidos en $A$.
- Un conjunto $A$ es un **superconjunto** de otro conjunto $B$ si todos los elementos de $B$ están incluidos en $A$.

Veamos un ejemplo con los siguientes conjuntos:

```
>>> A = {2, 4, 6, 8, 10}
>>> B = {4, 6, 8}
```

En Python podemos realizar comprobaciones de inclusión (subconjuntos y superconjuntos) utilizando operadores clásicos de comparación:

$B \subset A$

```
>>> B < A  # subconjunto
True
```

$B \subseteq A$

```
>>> B <= A
True
```

$A \supset B$

```
>>> A > B  # superconjunto
True
```

$A \supseteq B$

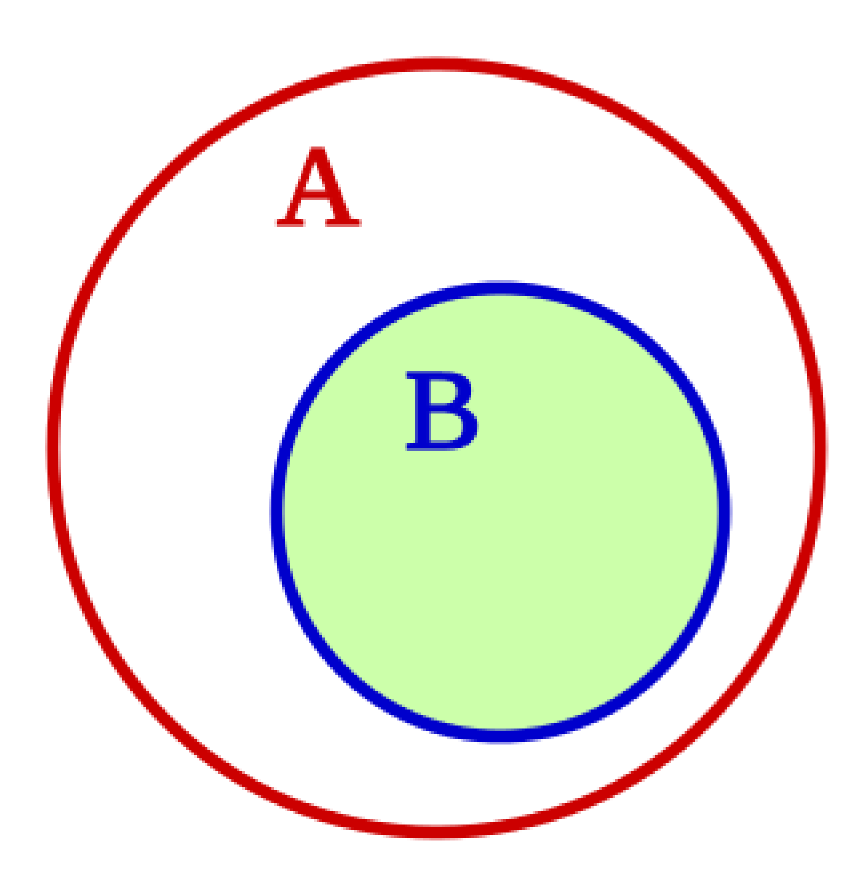



Figura 10: Subconjuntos y Superconjuntos

```
>>> B >= A
True
```

### 5.4.5 Conjuntos por comprensión

Los conjuntos, al igual que las *listas* y los *diccionarios*, también se pueden crear por comprensión.

Veamos un ejemplo en el que construimos un conjunto por comprensión con aquellos números enteros múltiplos de 3 en el rango $[0, 20)$:

```
>>> m3 = {number for number in range(0, 20) if number % 3 == 0}

>>> m3
{0, 3, 6, 9, 12, 15, 18}
```

**Ejercicio**

pycheck: **common_consonants**

### 5.4.6 Conjuntos inmutables

Python ofrece la posibilidad de crear **conjuntos inmutables** haciendo uso de la función `frozenset()` que recibe cualquier iterable como argumento.

Supongamos que recibimos una serie de calificaciones de exámenes y queremos crear un conjunto inmutable con los posibles niveles (categorías) de calificaciones:

```
>>> marks = [1, 3, 2, 3, 1, 4, 2, 4, 5, 2, 5, 5, 3, 1, 4]

>>> marks_levels = frozenset(marks)

>>> marks_levels
frozenset({1, 2, 3, 4, 5})
```

Veamos qué ocurre si intentamos modificar este conjunto:

```
>>> marks_levels.add(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

**Nota:** Los `frozenset` son a los `sets` lo que las tuplas a las listas: una forma de «congelar» los valores para que no se puedan modificar.

### AMPLIAR CONOCIMIENTOS

- Sets in Python

## 5.5 Ficheros



Aunque los ficheros encajarían más en un apartado de «*entrada/salida*» ya que representan un **medio de almacenamiento persistente**, también podrían ser vistos como *estructuras de datos*, puesto que nos permiten guardar la información y asignarles un cierto formato.[1]

Un **fichero** es un *conjunto de bytes* almacenados en algún *dispositivo*. El sistema de ficheros es la estructura lógica que alberga los ficheros y está jerarquizado a través de *directorios* (o carpetas). **Cada fichero se identifica unívocamente a través de una ruta que nos permite acceder a él**.

[1] Foto original de portada por Maksym Kaharlytskyi en Unsplash.

### 5.5.1 Lectura de un fichero

Python ofrece la función `open()` para «abrir» un fichero. Esta apertura se puede realizar en 3 modos distintos:

- **Lectura** del contenido de un fichero existente.
- **Escritura** del contenido en un fichero nuevo.
- **Añadido** al contenido de un fichero existente.

Veamos un ejemplo para leer el contenido de un fichero en el que se encuentran las temperaturas mínimas y máximas de cada día de la última semana. El fichero está en la subcarpeta (*ruta relativa*) `files/temps.dat` y tiene el siguiente contenido:

```
23 29
23 31
26 34
23 33
22 29
22 28
22 28
```

Lo primero será abrir el fichero:

```
>>> f = open('files/temps.dat', 'r')
```

La función `open()` recibe como primer argumento la **ruta al fichero** que queremos manejar (como un «string») y como segundo argumento el **modo de apertura** (también como un «string»). Nos **devuelve el manejador del fichero**, que en este caso lo estamos asignando a una variable llamada `f` pero le podríamos haber puesto cualquier otro nombre.

**Nota:** Es importante dominar los conceptos de **ruta relativa** y **ruta absoluta** para el trabajo con ficheros. Véase este artículo de DeNovatoANovato.

El **manejador del fichero** se implementa mediante un flujo de entrada/salida para las operaciones de lectura/escritura. Este objeto almacena, entre otras cosas, la *ruta al fichero*, el *modo de apertura* y la *codificación*:

```
>>> f
<_io.TextIOWrapper name='files/temps.dat' mode='r' encoding='UTF-8'>
```

**Truco:** Existen muchas codificaciones de caracteres para ficheros, pero la más utilizada es UTF-8 ya que es capaz de representar cualquier caracter Unicode al utilizar una longitud variable de 1 a 4 bytes.

Hay que tener en cuenta que la ruta al fichero que abrimos (*en modo lectura*) **debe existir**, ya que de lo contrario obtendremos un error:

```
>>> f = open('foo.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'foo.txt'
```

Una vez abierto el fichero ya podemos proceder a leer su contenido. Para ello Python nos ofrece la posibilidad de leer todo el fichero de una vez o bien leerlo línea a línea.

### Lectura completa de un fichero

Siguiendo con nuestro ejemplo de temperaturas, veamos cómo leer todo el contenido del fichero de una sola vez. Para esta operación, Python nos provee, al menos, de dos funciones:

**read()** Devuelve todo el contenido del fichero como una cadena de texto (`str`):

```
>>> # Podemos obviar 'r' ya que es el modo por defecto!
>>> f = open('files/temps.dat')

>>> f.read()
'23 29\n23 31\n26 34\n23 33\n22 29\n22 28\n22 28\n'
```

**readlines()** Devuelve todo el contenido del fichero como una lista (`list`) donde cada elemento es una línea:

```
>>> f = open('files/temps.dat')

>>> f.readlines()
['23 29\n', '23 31\n', '26 34\n', '23 33\n', '22 29\n', '22 28\n', '22 28\n']
```

**Importante:** Nótese que, en ambos casos, los saltos de línea `\n` siguen apareciendo en los datos leídos, por lo que habría que «limpiar» estos caracteres. Para ello se recomienda utilizar *las funciones ya vistas de cadenas de texto*.

### Lectura línea a línea

Hay situaciones en las que interesa leer el contenido del fichero línea a línea. Imaginemos un fichero de tamaño considerable (varios GB). Si intentamos leer completamente este fichero de sola una vez podríamos ocupar demasiada RAM y reducir el rendimiento de nuestra máquina.

Es por ello que Python nos ofrece varias aproximaciones a la lectura de ficheros línea a línea. La más usada es iterar sobre el propio *manejador* del fichero, ya que los ficheros son estructuras de datos **iterables**:

```
>>> f = open('files/temps.dat')

>>> for line in f:    # that easy!
...     print(line)
...
23 29

23 31

26 34

23 33

22 29

22 28

22 28
```

**Truco:** Igual que pasaba anteriormente, la lectura línea por línea también incluye el **salto de línea** `\n` lo que provoca un «doble espacio» entre cada una de las salidas. Bastaría con aplicar `line.strip()` para eliminarlo.

### Lectura de una línea

Hay ocasiones en las que nos interesa leer únicamente una sola línea. Es cierto que esto se puede conseguir mediante la aproximación anterior. Sería algo como:

```
>>> f = open('files/temps.dat')

>>> for line in f:
...     print(line)
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     break
...
23 29
```

Pero Python también ofrece la función `readline()` que nos devuelve la siguiente línea del fichero:

```
>>> f = open('files/temps.dat')

>>> f.readline()
'23 29\n'
```

Es importante señalar que cuando utilizamos la función `readline()` **el «puntero de lectura» se desplaza a la siguiente línea del fichero**, con lo que podemos seguir cargando la información según nos interese:

```
>>> f = open('files/temps.dat')

>>> # Lectura de las 3 primeras líneas
>>> for _ in range(3):
...     print(f.readline().strip())
...
23 29
23 31
26 34

>>> # Lectura de las restantes líneas (4)
>>> for line in f:
...     print(line.strip())
...
23 33
22 29
22 28
22 28
```

### Los ficheros se agotan

Hay que tener en cuenta que, una vez abierto el fichero, **la lectura de su contenido se puede realizar una única vez**. O dicho de otra manera, el iterable que lleva implícito «se agota».

Veamos este escenario con el ejemplo anterior:

```
>>> f = open('files/temps.dat')

>>> for line in f:
...     print(line.strip(), end=' ')
...
23 29 23 31 26 34 23 33 22 29 22 28 22 28

>>> for line in f:
...     print(line.strip(), end=' ')
... # No hay salida!!
```

Esto mismo ocurre si utilizamos funciones como `read()` o `readlines()`.

**Advertencia:** Por este motivo y también por cuestiones de legibilidad del código, deberíamos abrir un fichero una única vez y realizar todas las operaciones de lectura necesarias, siempre que las circunstancias lo permitan.

### 5.5.2 Escritura en un fichero

Para escribir texto en un fichero hay que abrir dicho fichero en **modo escritura**. Para ello utilizamos el *argumento adicional* en la función `open()` que indica esta operación:

```
>>> f = open('files/canary-iata.dat', 'w')
```

**Nota:** Si bien el fichero en sí mismo se crea al abrirlo en modo escritura, la **ruta** hasta ese fichero no. Eso quiere decir que debemos asegurarnos que **las carpetas hasta llegar a dicho fichero existen**. En otro caso obtenemos un error de tipo `FileNotFoundError`.

Ahora ya podemos hacer uso de la función `write()` para enviar contenido al fichero abierto.

Supongamos que queremos volcar el contenido de una lista/tupla en dicho fichero. En este caso partimos de los *códigos IATA* de aeropuertos de las Islas Canarias[2].

```
>>> canary_iata = ('TFN', 'TFS', 'LPA', 'GMZ', 'VDE', 'SPC', 'ACE', 'FUE')

>>> for code in canary_iata:
...     f.write(code + '\n')
...

>>> f.close()
```

[2] Fuente: Smart Drone

Nótese:

**Línea 4** Escritura de cada código en el fichero. La función `write()` no incluye el salto de línea por defecto, así que lo añadimos de *manera explícita*.

**Línea 7** Cierre del fichero con la función `close()`. Especialmente en el caso de la escritura de ficheros, se recomienda encarecidamente cerrar los ficheros para evitar pérdida de datos.

**Advertencia:** Siempre que se abre un fichero en **modo escritura** utilizando el argumento `'w'`, el fichero se inicializa, borrando cualquier contenido que pudiera tener.

Otra forma de **escribir la tupla «de una sola vez»** podría ser utilizando la función `join()` con el *salto de línea* como separador:

```
>>> canary_iata = ('TFN', 'TFS', 'LPA', 'GMZ', 'VDE', 'SPC', 'ACE', 'FUE')

>>> f = open('files/canary-iata.dat', 'w')

>>> f.write('\n'.join(canary_iata))

>>> f.close()
```

En el caso de que ya tengamos una **lista (iterable) cuyos elementos tengan el formato de salida que necesitamos** (incluyendo salto de línea si así fuera necesario) podemos utilizar la función `writelines()` que nos ofrece Python.

Siguiendo con el ejemplo anterior, imaginemos un escenario en el que la tupla ya contiene los saltos de línea:

```
>>> canary_iata = ('TFN\n', 'TFS\n', 'LPA\n', 'GMZ\n', 'VDE\n', 'SPC\n', 'ACE\n',
↪'FUE\n')

>>> f = open('files/canary-iata.dat', 'w')

>>> f.writelines(canary_iata)

>>> f.close()
```

**Truco:** Esta aproximación puede ser interesante cuando leemos de un fichero y escribimos en otro ya que las líneas «vienen» con el salto de línea ya incorporado.

### 5.5.3 Añadido a un fichero

La única diferencia entre añadir información a un fichero y *escribir información en un fichero* es el modo de apertura del fichero. En este caso utilizamos `'a'` por «append»:

```
>>> f = open('more-data.txt', 'a')
```

En este caso el fichero `more-data.txt` se abrirá en *modo añadir* con lo que las llamadas a la función `write()` hará que aparezcan nueva información al final del contenido ya existente en dicho fichero.

### 5.5.4 Usandos contextos

Python ofrece *gestores de contexto* como una solución para establecer reglas de entrada y salida a un determinado bloque de código.

En el caso que nos ocupa, usaremos la sentencia `with` y el contexto creado se ocupará de cerrar adecuadamente el fichero que hemos abierto, liberando así sus recursos:

```
1  >>> with open('files/temps.dat') as f:
2  ...     for line in f:
3  ...         min_temp, max_temp = line.strip().split()
4  ...         print(min_temp, max_temp)
5  ...
6  23 29
7  23 31
8  26 34
9  23 33
10 22 29
11 22 28
12 22 28
```

**Línea 1** Apertura del fichero en *modo lectura* utilizando el gestor de contexto definido por la palabra reservada `with`.

**Línea 2** Lectura del fichero línea a línea utilizando la iteración sobre el *manejador del fichero*.

**Línea 3** Limpieza de saltos de línea con `strip()` encadenando la función `split()` para separar las dos temperaturas por el carácter *espacio*. Ver *limpiar una cadena* y *dividir una cadena*.

**Línea 4** Imprimir por pantalla la temperatura mínima y la máxima.

---

**Nota:** Es una buena práctica usar `with` cuando se manejan ficheros. La ventaja es que el fichero se cierra adecuadamente en cualquier circunstancia, incluso si se produce cualquier

**tipo de error**.

Hay que prestar atención a la hora de escribir valores numéricos en un fichero, ya que el método `write()` por defecto espera ver un «string» como argumento:

```
>>> lottery = [43, 21, 99, 18, 37, 99]

>>> with open('files/lottery.dat', 'w') as f:
...     for number in lottery:
...         f.write(number + '\n')
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: write() argument must be str, not int
```

**Importante:** Para evitar este tipo de **errores**, se debe convertir a `str` aquellos valores que queramos usar con la función `write()` para escribir información en un fichero de texto. Los *f-strings* son tu aliado.

## EJERCICIOS DE REPASO

1. pycheck: **avg_temps**
2. pycheck: **wc**
3. pycheck: **read_csv**
4. pycheck: **txt2md**
5. pycheck: **find_words**
6. pycheck: **sum_matrix**
7. pycheck: **longest_word**
8. pycheck: **word_freq**
9. pycheck: **get_line**
10. pycheck: **replace_chars**
11. pycheck: **histogram**
12. pycheck: **submarine**

## AMPLIAR CONOCIMIENTOS

- Reading and Writing Files in Python
- Python Context Managers and the «with» Statement

# CAPÍTULO 6

# Modularidad

La **modularidad** es la característica de un sistema que permite que sea estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan solidariamente para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de **módulo**. Idealmente un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de entradas y salidas bien definidas.[1]

En este capítulo veremos las facilidades que nos proporciona Python para trabajar en la línea de modularidad del código.

[1] Definición de modularidad en Wikipedia

# 6.1 Funciones



El concepto de **función** es básico en prácticamente cualquier lenguaje de programación. Se trata de una estructura que nos permite agrupar código. Persigue dos objetivos claros:

1. **No repetir** fragmentos de código en un programa.
2. **Reutilizar** el código en distintos escenarios.

Una función viene *definida* por su *nombre*, sus *parámetros* y su *valor de retorno*. Esta parametrización de las funciones las convierten en una poderosa herramienta ajustable a las circunstancias que tengamos. Al *invocarla* estaremos solicitando su ejecución y obtendremos unos resultados.[1]

## 6.1.1 Definir una función

Para definir una función utilizamos la palabra reservada `def` seguida del **nombre**[6] de la función. A continuación aparecerán 0 o más **parámetros** separados por comas (entre paréntesis), finalizando la línea con **dos puntos** : En la siguiente línea empezaría el **cuerpo** de la función que puede contener 1 o más **sentencias**, incluyendo (o no) una **sentencia de retorno** con el resultado mediante `return`.

[1] Foto original por Nathan Dumlao en Unsplash.

[6] Las *reglas aplicadas a nombres de variables* también se aplican a nombres de funciones.

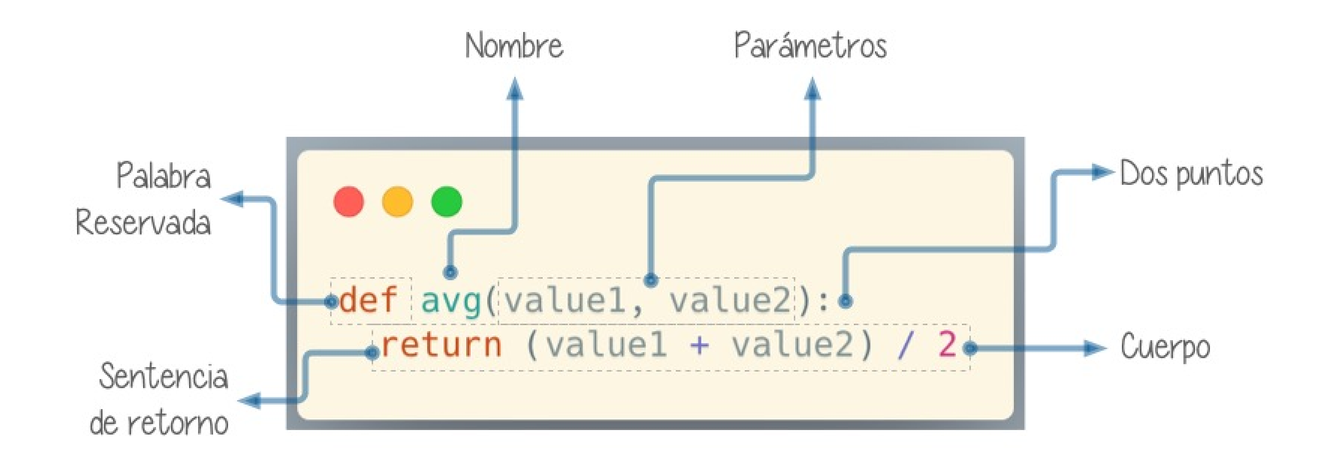



Figura 1: Definición de una función en Python

**Advertencia:** Prestar especial atención a los dos puntos : porque suelen olvidarse en la *definición de la función*.

Hagamos una primera función sencilla que no recibe parámetros:

```
def say_hello():
    print('Hello!')
```

**Nota:** Nótese la *indentación* (sangrado) del *cuerpo* de la función.

Los **nombres de las funciones** siguen *las mismas reglas que las variables* y, como norma general, se suelen utilizar **verbos en infinitivo** para su definición: `load_data`, `store_values`, `reset_cart`, `filter_results`, `block_request`, …

### Invocar una función

Para invocar (o «llamar») a una función sólo tendremos que escribir su nombre seguido de paréntesis. En el caso de la función sencilla (vista anteriormente) se haría así:

```
>>> def say_hello():
...     print('Hello!')
...

>>> say_hello()
Hello!
```

**Nota:** Como era de esperar, al invocar a esta función obtenemos un mensaje por pantalla, fruto de la ejecución del cuerpo de la función.

Cuando queremos **invocar a una función dentro de un fichero** `*.py` lo haremos del mismo modo que hemos visto en el intérprete interactivo:

```
def say_hello():
    print('Hello!')

# Llamada a la función (primer nivel de indentación)
say_hello()
```

### Retornar un valor

Las funciones pueden retornar (o «devolver») un valor. Veamos un ejemplo muy sencillo:

```
>>> def one():
...     return 1
...

>>> one()
1
```

**Importante:** No confundir `return` con `print()`. El valor de retorno de una función nos permite usarlo fuera de su contexto. El hecho de añadir `print()` al cuerpo de una función es algo «coyuntural» y no modifica el resultado de la lógica interna.

**Nota:** En la sentencia `return` podemos incluir variables y expresiones, no únicamente literales.

Pero no sólo podemos invocar a la función directamente, también la podemos integrar en otras expresiones. Por ejemplo en condicionales:

```
>>> if one() == 1:
...     print('It works!')
... else:
...     print('Something is broken')
...
It works!
```

Si una función no incluye un `return` de forma explícita, devolverá *None* de forma implícita:

```
>>> def empty():
...     x = 0
...     # return None

>>> print(empty())
None
```

Existe la posibilidad de usar la sentencia `return` «a secas» (que también devuelve `None`) y hace que «salgamos» inmediatamente de la función:

```
>>> def quick():
...     return
...

>>> print(quick())
None
```

**Advertencia:** En general, esto **no se considera una buena práctica** salvo que sepamos lo que estamos haciendo. Si la función debe devolver `None` es preferible ser **explícito** y utilizar `return None`. Aunque es posible que en ciertos escenarios nos interese dicha aproximación.

### Retornando múltiples valores

Una función puede retornar más de un valor. El «secreto» es hacerlo **mediante una tupla**:

```
>>> def multiple():
...     return 0, 1  # es una tupla!
...
```

Veamos qué ocurre si invocamos a esta función:

```
>>> result = multiple()

>>> result
(0, 1)

>>> type(result)
tuple
```

Por lo tanto, podremos aplicar el *desempaquetado de tuplas* sobre el valor retornado por la función:

```
>>> a, b = multiple()

>>> a
0

>>> b
1
```

### 6.1.2 Parámetros y argumentos

Si una función no dispusiera de valores de entrada estaría muy limitada en su actuación. Es por ello que los **parámetros** nos permiten variar los datos que consume una función para obtener distintos resultados. Vamos a empezar a crear funciones que reciben **parámetros**.

En este caso escribiremos una función que recibe un valor numérico y devuelve su raíz cuadrada:

```
>>> def sqrt(value):
...     return value ** (1/2)
...

>>> sqrt(4)
2.0
```

**Nota:** En este caso, el valor `4` es un **argumento** de la función.

Cuando llamamos a una función con *argumentos*, los valores de estos argumentos se copian en los correspondientes *parámetros* dentro de la función:

**Truco:** La sentencia `pass` permite «no hacer nada». Es una especie de «*placeholder*».

Veamos otra función con dos parámetros y algo más de lógica de negocio:[2]

```
>>> def _min(a, b):
...     if a < b:
...         return a
...     else:
...         return b
...
```

(continué en la próxima página)

[2] Término para identificar el «algoritmo» o secuencia de instrucciones derivadas del procesamiento que corresponda.

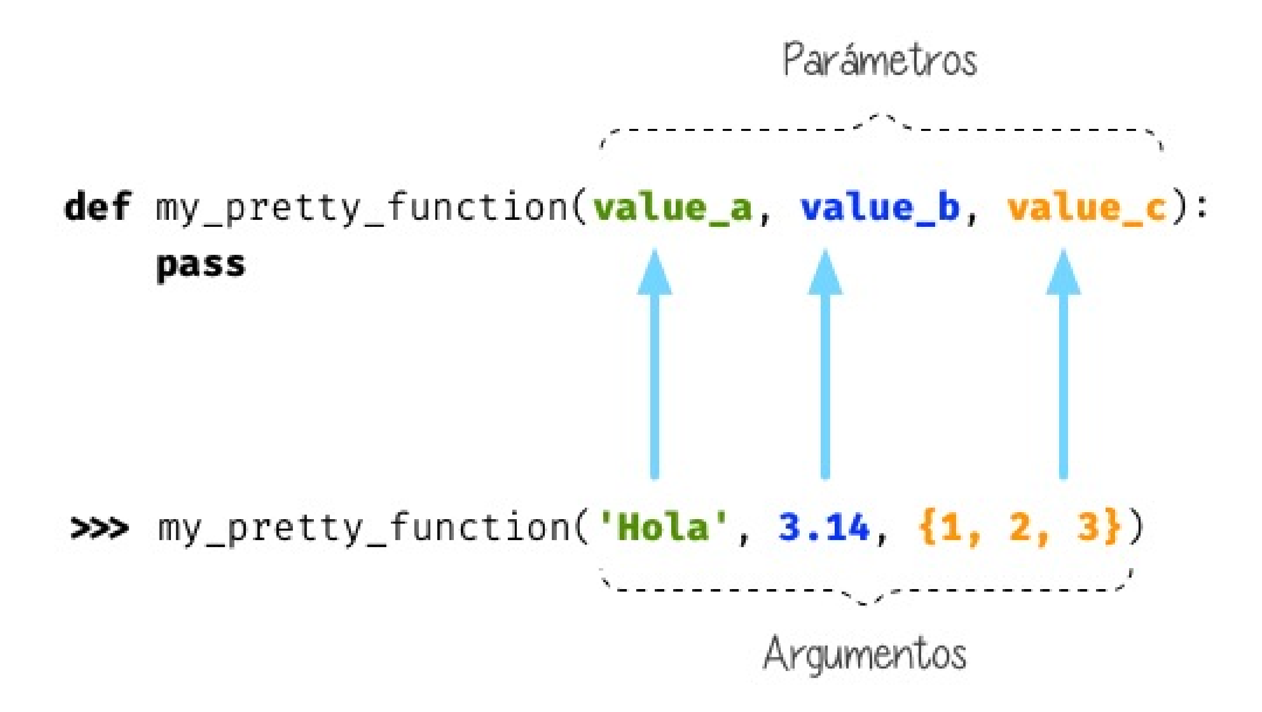



Figura 2: Parámetros y argumentos de una función

(proviene de la página anterior)

```
>>> _min(7, 9)
7
```

Nótese que la sentencia `return` puede escribirse en **múltiples ocasiones** y puede encontrarse en **cualquier lugar** de la función, no necesariamente al final del cuerpo. Esta técnica puede ser beneficiosa en múltiples escenarios.

Uno de esos escenarios se relaciona con el concepto de **cláusula guarda**: una pieza de código que normalmente está al comienzo de la función y comprueba una serie de condiciones para continuar con la ejecución o cortarla[10].

Teniendo en cuenta que la sentencia `return` finaliza la ejecución de una función, es viable **eliminar la sentencia** `else` del ejemplo visto anteriormente:

```
>>> def _min(a, b):
...     if a < b:
...         return a
...     return b
```

(continué en la próxima página)

[10] Para más información sobre las cláusulas guarda, véase este artículo de Miguel G. Flores

(proviene de la página anterior)

```
>>> _min(7, 9)
7
```

**Ejercicio**

pycheck: **squared_sum**

## Argumentos posicionales

Los **argumentos posicionales** son aquellos argumentos que se copian en sus correspondientes parámetros **en orden**.

Vamos a mostrar un ejemplo definiendo una función que construye una «cpu» a partir de 3 parámetros:

```
>>> def build_cpu(vendor, num_cores, freq):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
...     )
...
```

Una posible llamada a la función con argumentos posicionales sería la siguiente:

```
>>> build_cpu('AMD', 8, 2.7)
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Lo que ha sucedido es un **mapeo** directo entre argumentos y parámetros en el mismo orden que estaban definidos:

| Parámetro | Argumento |
|---|---|
| vendor | AMD |
| num_cores | 8 |
| freq | 2.7 |

Pero es evidente que una clara desventaja del uso de argumentos posicionales es que se necesita **recordar el orden** de los argumentos. Un error en la posición de los argumentos puede generar resultados indeseados:

```
>>> build_cpu(8, 2.7, 'AMD')
{'vendor': 8, 'num_cores': 2.7, 'freq': 'AMD'}
```

### Argumentos nominales

En esta aproximación los argumentos no son copiados en un orden específico sino que **se asignan por nombre a cada parámetro**. Ello nos permite evitar el problema de conocer cuál es el orden de los parámetros en la definición de la función. Para utilizarlo, basta con realizar una asignación de cada argumento en la propia llamada a la función.

Veamos la misma llamada que hemos hecho en el ejemplo de construcción de la «cpu» pero ahora utilizando paso de argumentos nominales:

```
>>> build_cpu(vendor='AMD', num_cores=8, freq=2.7)
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Se puede ver claramente que el orden de los argumentos no influye en el resultado final:

```
>>> build_cpu(num_cores=8, freq=2.7, vendor='AMD')
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

### Argumentos posicionales y nominales

Python permite **mezclar argumentos posicionales y nominales** en la llamada a una función:

```
>>> build_cpu('INTEL', num_cores=4, freq=3.1)
{'vendor': 'INTEL', 'num_cores': 4, 'freq': 3.1}
```

Pero hay que tener en cuenta que, en este escenario, **los argumentos posicionales siempre deben ir antes** que los argumentos nominales. Esto tiene mucho sentido ya que, de no hacerlo así, Python no tendría forma de discernir a qué parámetro corresponde cada argumento:

```
>>> build_cpu(num_cores=4, 'INTEL', freq=3.1)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

## Argumentos mutables e inmutables

### Nivel intermedio

Cuando realizamos modificaciones a los argumentos de una función es importante tener en cuenta si son **mutables** (listas, diccionarios, conjuntos, …) o **inmutables** (tuplas, enteros, flotantes, cadenas de texto, …) ya que podríamos obtener efectos colaterales no deseados.

Supongamos que nos piden escribir una función que reciba una lista y que devuelva sus valores elevados al cuadrado. Pero lo hacemos «malamente»:

```
>>> values = [2, 3, 4]

>>> def square_it(values):
...     # NO HAGAS ESTO
...     for i in range(len(values)):
...         values[i] **= 2
...     return values

>>> square_it(values)
[4, 9, 16]

>>> values  # ???
[4, 9, 16]
```

> **Advertencia:** Esto **no es una buena práctica**. O bien documentar que el argumento puede modificarse o bien retornar un nuevo valor. Por regla general, no se recomienda que las funciones modifiquen argumentos de entrada, salvo que sea específicamente lo que estamos buscando.

## Parámetros por defecto

Es posible especificar **valores por defecto** en los parámetros de una función. En el caso de que no se proporcione un valor al argumento en la llamada a la función, el parámetro correspondiente tomará el valor definido por defecto.

Siguiendo con el ejemplo de la «cpu», podemos asignar *2.0GHz* como frecuencia por defecto. La definición de la función cambiaría ligeramente:

```
>>> def build_cpu(vendor, num_cores, freq=2.0):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     )
...
```

Llamada a la función sin especificar frecuencia de «cpu»:

```
>>> build_cpu('INTEL', 2)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 2.0}
```

Llamada a la función indicando una frecuencia concreta de «cpu»:

```
>>> build_cpu('INTEL', 2, 3.4)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 3.4}
```

### Nivel intermedio

Es importante tener presente que los valores por defecto en los parámetros se calculan cuando se **define** la función, no cuando se **ejecuta**. Veamos un ejemplo siguiendo con el caso anterior:

```
>>> DEFAULT_FREQ = 2.0

>>> def build_cpu(vendor, num_cores, freq=DEFAULT_FREQ):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
...     )
...

>>> build_cpu('AMD', 4)
{'vendor': 'AMD', 'num_cores': 4, 'freq': 2.0}

>>> DEFAULT_FREQ = 3.5

>>> build_cpu('AMD', 4)
{'vendor': 'AMD', 'num_cores': 4, 'freq': 2.0}
```

---

### Ejercicio

pycheck: **factorial**

---

## Modificando parámetros mutables

### Nivel avanzado

Hay que tener cuidado a la hora de manejar los parámetros que pasamos a una función ya que *podemos obtener resultados indeseados*, especialmente cuando trabajamos con *tipos de datos mutables*.

Supongamos una función que añade elementos a una lista que pasamos por parámetro. La idea es que si no pasamos la lista, ésta siempre empiece siendo vacía. Hagamos una serie de pruebas pasando alguna lista como segundo argumento:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...

>>> buggy('a', [])
['a']

>>> buggy('b', [])
['b']

>>> buggy('a', ['x', 'y', 'z'])
['x', 'y', 'z', 'a']

>>> buggy('b', ['x', 'y', 'z'])
['x', 'y', 'z', 'b']
```

Aparentemente todo está funcionando de manera correcta, pero veamos qué ocurre en las siguientes llamadas:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...

>>> buggy('a')
['a']

>>> buggy('b')  # Se esperaría ['b']
['a', 'b']
```

Obviamente algo no ha funcionado correctamente. Se esperaría que `result` tuviera una lista vacía en cada ejecución. Sin embargo esto no sucede por estas dos razones:

1. El valor por defecto se establece cuando se define la función.

2. La variable `result` apunta a una zona de memoria en la que se modifican sus valores.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/sBNpVT2

A riesgo de perder el *parámetro por defecto*, una posible solución sería la siguiente:

```
>>> def works(arg):
...     result = []
...     result.append(arg)
...     return result
...

>>> works('a')
['a']

>>> works('b')
['b']
```

La forma de arreglar el código anterior utilizando un parámetro con valor por defecto sería utilizar un **tipo de dato inmutable** y tener en cuenta cuál es la primera llamada:

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...

>>> nonbuggy('a')
['a']

>>> nonbuggy('b')
['b']

>>> nonbuggy('a', ['x', 'y', 'z'])
['x', 'y', 'z', 'a']

>>> nonbuggy('b', ['x', 'y', 'z'])
['x', 'y', 'z', 'b']
```

### Empaquetar/Desempaquetar argumentos

**Nivel intermedio**

Python nos ofrece la posibilidad de empaquetar y desempaquetar argumentos cuando estamos invocando a una función, tanto para **argumentos posicionales** como para **argumentos nominales**.

Y de esto se deriva el hecho de que podamos utilizar un **número variable de argumentos** en una función, algo que puede ser muy interesante según el caso de uso que tengamos.

#### Empaquetar/Desempaquetar argumentos posicionales

Si utilizamos el operador `*` delante del nombre de un parámetro posicional, estaremos indicando que los argumentos pasados a la función se empaqueten en una **tupla**.

Veamos un ejemplo en el que vamos a **implementar una función para sumar un número variable de valores**. La función que tenemos disponible en Python no cubre este caso:

```
>>> sum(4, 3, 2, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() takes at most 2 arguments (4 given)
```

Para superar esta «limitación» vamos a hacer uso del `*` para empaquetar los argumentos posicionales:

```
>>> def _sum(*values: int) -> int:
...     print(f'{values=}')
...     result = 0
...     for value in values:  # values es una tupla
...         result += value
...     return result
...

>>> _sum(4, 3, 2, 1)
values=(4, 3, 2, 1)
10
```

Existe la posibilidad de usar el asterisco `*` en la llamada a la función para **desempaquetar** los argumentos posicionales:

```
>>> values = (4, 3, 2, 1)

>>> _sum(values)
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in _sum
TypeError: unsupported operand type(s) for +=: 'int' and 'tuple'

>>> # Desempaquetado: _sum(4, 3, 2, 1)
>>> _sum(*values)
values=(4, 3, 2, 1)
10
```

### Empaquetar/Desempaquetar argumentos nominales

Si utilizamos el operador `**` delante del nombre de un parámetro nominal, estaremos indicando que los argumentos pasados a la función se empaqueten en un **diccionario**.

Supongamos un ejemplo en el que queremos **encontrar la persona con mayor calificación de un examen**. Haremos uso del `**` para empaquetar los argumentos nominales:

```
>>> def best_student(**marks: int) -> str:
...     print(f'{marks=}')
...     max_mark = -1
...     for student, mark in marks.items():  # marks es un diccionario
...         if mark > max_mark:
...             max_mark = mark
...             best_student = student
...     return best_student
...

>>> best_student(ana=8, antonio=6, inma=9, javier=7)
marks={'ana': 8, 'antonio': 6, 'inma': 9, 'javier': 7}
'inma'
```

Al igual que veíamos previamente, existe la posibilidad de usar doble asterisco `**` en la llamada a la función para **desempaquetar** los argumentos nominales:

```
>>> marks = dict(ana=8, antonio=6, inma=9, javier=7)

>>> best_student(marks)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: best_student() takes 0 positional arguments but 1 was given

>>> # Desempaquetado: best_student(ana=8, antonio=6, inma=9, javier=7)
>>> best_student(**marks)
```

(continué en la próxima página)

(proviene de la página anterior)

```
marks={'ana': 8, 'antonio': 6, 'inma': 9, 'javier': 7}
'inma'
```

### Convenciones

En muchas ocasiones se utiliza `args` como nombre de parámetro para argumentos posicionales y `kwargs` como nombre de parámetro para argumentos nominales. Esto son únicamente **convenciones**, no hay obligación de utilizar estos nombres. Así, podemos encontrar funciones definidas de la siguiente manera:

```
>>> def func(*args, **kwargs):
...     # TODO
...     pass
...
```

### Forzando modo de paso de argumentos

Si bien Python nos da flexibilidad para pasar argumentos a nuestras funciones en modo nominal o posicional, existen opciones para forzar que dicho paso sea obligatorio para una determinada modalidad.

### Argumentos sólo nominales

**Nivel avanzado**

A partir de Python 3.0 se ofrece la posibilidad de obligar a que determinados parámetros de la función sean pasados sólo por nombre.

Para ello, en la definición de los parámetros de la función, tendremos que incluir un parámetro especial `*` que delimitará el tipo de parámetros. Así, todos los parámetros a la derecha del separador estarán **obligados** a ser nominales:

Ejemplo:

```
>>> def sum_power(a, b, *, power=False):
...     if power:
...         a **= 2
...         b **= 2
...     return a + b
...

>>> sum_power(3, 4)
```

(continué en la próxima página)




Figura 3: Separador para especificar parámetros sólo nominales

(proviene de la página anterior)

```
7

>>> sum_power(a=3, b=4)
7

>>> sum_power(3, 4, power=True)
25

>>> sum_power(3, 4, True)
---------------------------------------------------------------------------
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum_power() takes 2 positional arguments but 3 were given
```

### Argumentos sólo posicionales

#### Nivel avanzado

A partir de Python 3.8 se ofrece la posibilidad de obligar a que determinados parámetros de la función sean pasados sólo por posición.

Para ello, en la definición de los parámetros de la función, tendremos que incluir un parámetro especial `/` que delimitará el tipo de parámetros. Así, todos los parámetros a la izquierda del delimitador estarán **obligados** a ser posicionales:

Ejemplo:

```
>>> def sum_power(a, b, /, power=False):
...     if power:
...         a **= 2
```

(continué en la próxima página)

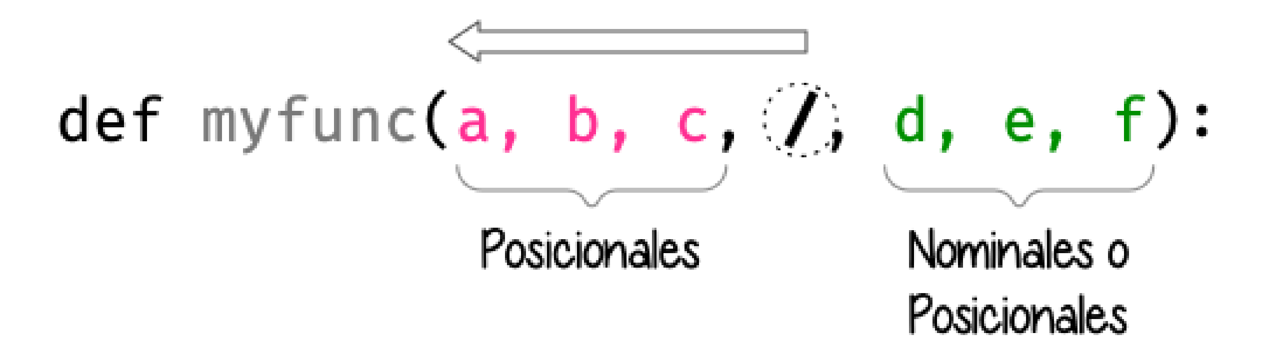



Figura 4: Separador para especificar parámetros sólo posicionales

(proviene de la página anterior)

```
...         b **= 2
...     return a + b
...

>>> sum_power(3, 4)
7

>>> sum_power(3, 4, True)
25

>>> sum_power(3, 4, power=True)
25

>>> sum_power(a=3, b=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum_power() got some positional-only arguments passed as keyword␣
↪arguments: 'a, b'
```

### Fijando argumentos posicionales y nominales

Si mezclamos las dos estrategias anteriores podemos forzar a que una función reciba argumentos de un modo concreto.

Continuando con el ejemplo anterior, podríamos hacer lo siguiente:

```
>>> def sum_power(a, b, /, *, power=False):
...     if power:
...         a **= 2
```

(continúe en la próxima página)

(proviene de la página anterior)

```
...         b **= 2
...     return a + b
...

>>> sum_power(3, 4, power=True)  # Único modo posible de llamada
25
```

**Ejercicio**

pycheck: **consecutive_freqs**

## Funciones como parámetros

### Nivel avanzado

Las funciones se pueden utilizar en cualquier contexto de nuestro programa. Son objetos que pueden ser asignados a variables, usados en expresiones, devueltos como valores de retorno o pasados como argumentos a otras funciones.

Veamos un primer ejemplo en el que pasamos una función como argumento:

```
>>> def success():
...     print('Yeah!')
...

>>> type(success)
function

>>> def doit(f):
...     f()
...

>>> doit(success)
Yeah!
```

Veamos un segundo ejemplo en el que pasamos, no sólo una función como argumento, sino los valores con los que debe operar:

```
>>> def repeat_please(text, times=1):
...     return text * times
...

>>> type(repeat_please)
```

(continué en la próxima página)

(proviene de la página anterior)

```
function

>>> def doit(f, arg1, arg2):
...     return f(arg1, arg2)
...

>>> doit(repeat_please, 'Functions as params', 2)
'Functions as paramsFunctions as params'
```

### 6.1.3 Documentación

Ya hemos visto que en Python podemos incluir *comentarios* para explicar mejor determinadas zonas de nuestro código.

Del mismo modo podemos (y en muchos casos **debemos**) adjuntar **documentación** a la definición de una función incluyendo una cadena de texto (**docstring**) al comienzo de su cuerpo:

```
>>> def sqrt(value):
...     'Returns the square root of the value'
...     return value ** (1/2)
...
```

La forma más ortodoxa de escribir un `docstring` es utilizando *triples comillas*:

```
>>> def closest_int(value):
...     '''Returns the closest integer to the given value.
...     The operation is:
...         1. Compute distance to floor.
...         2. If distance less than a half, return floor.
...            Otherwise, return ceil.
...     '''
...     floor = int(value)
...     if value - floor < 0.5:
...         return floor
...     else:
...         return floor + 1
...
```

Para ver el `docstring` de una función, basta con utilizar `help`:

```
>>> help(closest_int)

Help on function closest_int in module __main__:
```

(continué en la próxima página)

(proviene de la página anterior)

```
closest_int(value)
    Returns the closest integer to the given value.
    The operation is:
        1. Compute distance to floor.
        2. If distance less than a half, return floor.
           Otherwise, return ceil.
```

También es posible extraer información usando el símbolo de interrogación:

```
>>> closest_int?
Signature: closest_int(value)
Docstring:
Returns the closest integer to the given value.
The operation is:
    1. Compute distance to floor.
    2. If distance less than a half, return floor.
    Otherwise, return ceil.
File:      ~/aprendepython/<ipython-input-75-5dc166360da1>
Type:      function
```

**Importante:** Esto no sólo se aplica a funciones propias, sino a cualquier otra función definida en el lenguaje.

**Nota:** Si queremos ver el `docstring` de una función en «crudo» (sin formatear), podemos usar `<function>.__doc__`.

### Explicación de parámetros

Como ya se ha visto, es posible documentar una función utilizando un `docstring`. Pero la redacción y el formato de esta cadena de texto puede ser muy variada. Existen distintas formas de documentar una función (u otros objetos)[3]:

**reStructuredText docstrings** Formato de documentación recomendado por Python.

**Google docstrings** Formato de documentación recomendado por Google.

**NumPy-SciPy docstrings** Combinación de formatos reStructuredText y Google (usados por el proyecto NumPy).

**Epytext docstrings** Formato utilizado por Epydoc (una adaptación de Javadoc).

[3] Véase Docstring Formats.

Aunque cada uno tienes sus particularidades, todos comparten una misma estructura:

- Una primera línea de **descripción de la función**.
- A continuación especificamos las características de los **parámetros** (incluyendo sus tipos).
- Por último, indicamos si la función **retorna un valor** y sus características.

Aunque todos los formatos son válidos, nos centraremos en **reStructuredText** por ser el estándar propuesto por Python para la documentación.

**Ver también:**

*Google docstrings* y *Numpy docstrings* también son ampliamente utilizados, lo único es que necesitan de un módulo externo denominado Napoleon para que se puedan incluir en la documentación *Sphinx*.

### Sphinx

Sphinx es una herramienta para generar documentación usando el lenguaje reStructuredText o RST. Incluye un módulo «built-in» denominado autodoc el cual permite la autogeneración de documentación a partir de los «docstrings» definidos en el código.

Veamos el uso de este formato en la documentación de la siguiente función:

```
>>> def my_power(x, n):
...     '''Calculate x raised to the power of n.
...
...     :param x: number representing the base of the operation
...     :type x: int
...     :param n: number representing the exponent of the operation
...     :type n: int
...
...     :return: :math:`x^n`
...     :rtype: int
...     '''
...     result = 1
...     for _ in range(n):
...         result *= x
...     return result
...
```

Dentro del «docstring» podemos escribir con sintaxis reStructuredText – véase por ejemplo la expresión matemática en el tag `:return:` – lo que nos proporciona una gran flexibilidad.

---

**Nota:** La plataforma Read the Docs aloja la documentación de gran cantidad de proyectos. En muchos de los casos se han usado «docstrings» con el formato Sphinx visto anteriormente.

Un ejemplo de ello es la popular librería de Python requests.

## Anotación de tipos

### Nivel intermedio

Las anotaciones de tipos o **type-hints**[5] se introdujeron en Python 3.5 y permiten indicar tipos para los parámetros de una función y/o para su valor de retorno (*aunque también funcionan en creación de variables*).

Veamos un ejemplo en el que creamos una función para dividir una cadena de texto por la posición especificada en el parámetro:

```
>>> def ssplit(text: str, split_pos: int) -> tuple:
...     return text[:split_pos], text[split_pos:]
...

>>> ssplit('Always remember us this way', 15)
('Always remember', ' us this way')
```

Como se puede observar, vamos añadiendo los tipos después de cada parámetro utilizando `:` como separador. En el caso del valor de retorno usamos la flecha `->`

Quizás la siguiente ejecución pueda sorprender:

```
>>> ssplit([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 5)
([1, 2, 3, 4, 5], [6, 7, 8, 9, 10])
```

Efectivamente como habrás visto, **no hemos obtenido ningún error**, a pesar de que estamos pasando como primer argumento una lista en vez de una cadena de texto. Esto ocurre porque lo que hemos definido es simplemente una anotación de tipo, no una declaración de tipo. Existen herramientas como mypy que sí se encarga de comprobar este escenario.

## Valores por defecto

Al igual que ocurre en la definición ordinaria de funciones, cuando usamos anotaciones de tipos también podemos indicar un valor por defecto para los parámetros.

Veamos la forma de hacerlo continuando con el ejemplo anterior:

```
>>> def ssplit(text: str, split_pos: int = -1) -> tuple:
...     if split_pos == -1:
...         split_pos = len(text) // 2
```

(continué en la próxima página)

[5] Conocidos como «type hints» en terminología inglesa.

(proviene de la página anterior)

```
...     return text[:split_pos], text[split_pos:]
...

>>> ssplit('Always remember us this way')
('Always rememb', 'er us this way')
```

Simplemente añadimos el valor por defecto después de indicar el tipo.

Las **anotaciones de tipos** son una herramienta muy potente y que, usada de forma adecuada, permite complementar la documentación de nuestro código y aclarar ciertos aspectos, que a priori, pueden parecer confusos. Su aplicación estará en función de la necesidad detectada por parte del equipo de desarrollo.

### Tipos compuestos

Hay escenarios en los que necesitamos más expresividad de cara a la anotación de tipos. ¿Qué ocurre si queremos indicar una *lista de cadenas de texto* o un *conjunto de enteros*?

Veamos algunos ejemplos válidos:

| Anotación | Ejemplo |
|---|---|
| `list[str]` | `['A', 'B', 'C']` |
| `set[int]` | `{4, 3, 9}` |
| `dict[str, float]` | `{'x': 3.786, 'y': 2.198, 'z': 4.954}` |
| `tuple[str, int]` | `('Hello', 10)` |
| `tuple[float, ...]` | `(1.23, 5.21, 3.62) o (4.31, 6.87) o (7.11,)` |

### Múltiples tipos

En el caso de que queramos indicar que un determinado parámetro puede ser de un tipo o de otro hay que especificarlo utilizando el operador[7] `|`.

Veamos algunos ejemplos válidos:

| Anotación | Significado |
|---|---|
| `tuple□dict` | Tupla o diccionario |
| `list[str□int]` | Lista de cadenas de texto y/o enteros |
| `set[int□float]` | Conjunto de enteros y/o flotantes |

**Ver también:**

[7] Disponible a partir de Python 3.10.

Guía rápida para de anotación de tipos (mypy)

---

**Ejercicio**

pycheck: **mcount**

---

### Número indefinido

Cuando trabajamos con **parámetros que representan un número indefinido de valores**, las anotaciones de tipo sólo hacen referencia al tipo que contiene la tupla, no es necesario indicar que es una tupla.

En el siguiente ejemplo hay una función que calcula el máximo de una serie de valores enteros o flotantes, pero no indicamos que se reciben como tupla:

```
>>> def _max(*args: int | float):
...     ...
...
```

## 6.1.4 Tipos de funciones

**Nivel avanzado**

### Funciones anónimas «lambda»

**Una función lambda tiene las siguientes propiedades:**

1. Se escribe en una única sentencia (línea).
2. No tiene nombre (anónima).
3. Su cuerpo conlleva un `return` implícito.
4. Puede recibir cualquier número de parámetros.

Veamos un primer ejemplo de función «lambda» que nos permite **contar el número de palabras en una cadena de texto** dada. La **transformación de su versión clásica en su versión anónima** sería la siguiente:

> **Prudencia:** Aunque en muchas ocasiones se suelen «abreviar» los nombres de las variables en una función «lambda» no es obligatorio, y en muchos casos, puede que sea hasta contraproducente.

```
def num_words(text: str) -> int:
    return len(text.split())

lambda t: len(t.split())
```

Figura 5: Transformación en función «lambda»

A continuación probamos el comportamiento de la función anónima «lambda»:

```
>>> num_words = lambda t: len(t.split())

>>> type(num_words)
function

>>> num_words
<function __main__.<lambda>(t)>

>>> num_words('hola socio vamos a ver')
5
```

Veamos otro ejemplo en el que mostramos una tabla con el resultado de aplicar el «and» lógico mediante una función «lambda» que ahora recibe dos parámetros:

```
>>> logic_and = lambda x, y: x & y

>>> for i in range(2):
...     for j in range(2):
...         print(f'{i} & {j} = {logic_and(i, j)}')
...
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

### Lambdas como argumentos

Las funciones «lambda» son bastante utilizadas **como argumentos a otras funciones**. Un ejemplo claro de ello es la función `sorted` que recibe un parámetro opcional `key` donde se define la clave de ordenación.

Veamos cómo usar una función anónima «lambda» para ordenar una tupla de pares *longitud-latitud*:

```
>>> geoloc = (
... (15.623037, 13.258358),
... (55.147488, -2.667338),
... (54.572062, -73.285171),
... (3.152857, 115.327724),
... (-40.454262, 172.318877)
)

>>> # Ordenación por longitud (primer elemento de la tupla)
>>> sorted(geoloc)
[(-40.454262, 172.318877),
 (3.152857, 115.327724),
 (15.623037, 13.258358),
 (54.572062, -73.285171),
 (55.147488, -2.667338)]

>>> # Ordenación por latitud (segundo elemento de la tupla)
>>> sorted(geoloc, key=lambda t: t[1])
[(54.572062, -73.285171),
 (55.147488, -2.667338),
 (15.623037, 13.258358),
 (3.152857, 115.327724),
 (-40.454262, 172.318877)]
```

**Ejercicio**

pycheck: **sort_ages**

## Enfoque funcional

Como se comentó en la *introducción*, Python es un lenguaje de programación multiparadigma. Uno de los paradigmas menos explotados en este lenguaje es la **programación funcional**[4].

Python nos ofrece 3 funciones que encajan verdaderamente bien en este enfoque: `map()`, `filter()` y `reduce()`.

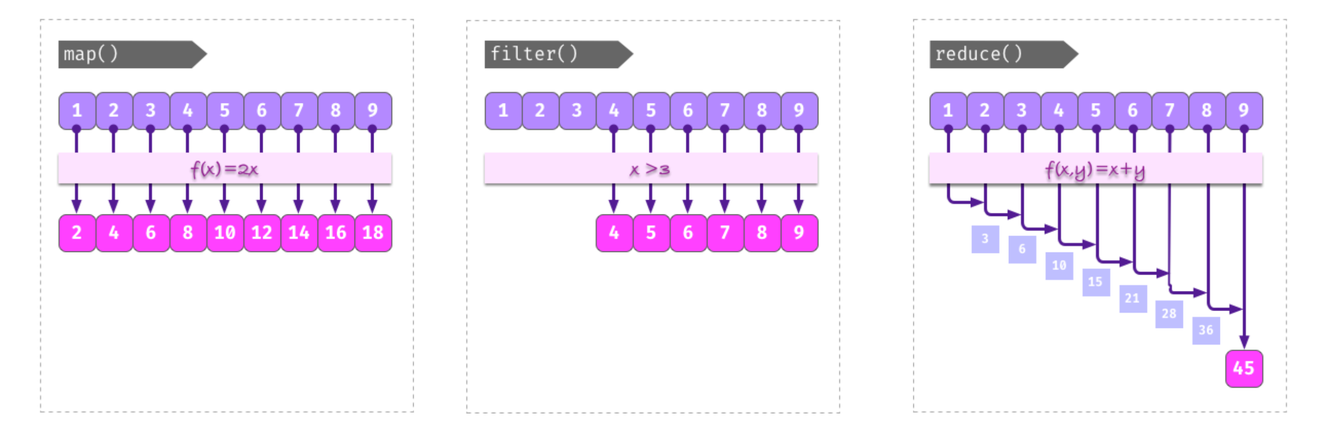



Figura 6: Rutinas muy enfocadas a programación funcional

### `map()`

Esta función **aplica otra función** sobre cada elemento de un iterable. Supongamos que queremos aplicar la siguiente función:

$$f(x) = \frac{x^2}{2} \qquad \forall x \in [1, 10]$$

```
>>> def f(x):
...     return x**2 / 2
...

>>> data = range(1, 11)

>>> map_gen = map(f, data)

>>> type(map_gen)
map

>>> list(map_gen)
[0.5, 2.0, 4.5, 8.0, 12.5, 18.0, 24.5, 32.0, 40.5, 50.0]
```

[4] Definición de *Programación funcional* en Wikipedia.

**Truco:** Hay que tener en cuenta que `map()` devuelve un *generador*, no directamente una lista.

Podemos obtener el mismo resultado aplicando una *función anónima «lambda»*:

```
>>> list(map(lambda x: x**2 / 2, data))
[0.5, 2.0, 4.5, 8.0, 12.5, 18.0, 24.5, 32.0, 40.5, 50.0]
```

En Python es posible «simular» un `map()` a través de una *lista por comprensión*:

```
>>> [x**2 / 2 for x in data]
[0.5, 2.0, 4.5, 8.0, 12.5, 18.0, 24.5, 32.0, 40.5, 50.0]
```

### `filter()`

Esta función **selecciona** aquellos elementos de un iterable que cumplan una determinada condición. Supongamos que queremos seleccionar sólo aquellos números impares dentro de un rango:

```
>>> def odd_number(x):
...     return x % 2 == 1
...

>>> data = range(1, 21)

>>> filter_gen = filter(odd_number, data)

>>> type(filter_gen)
filter

>>> list(filter_gen)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

**Truco:** Hay que tener en cuenta que `filter()` devuelve un *generador*, no directamente una lista.

Podemos obtener el mismo resultado aplicando una *función anónima «lambda»*:

```
>>> list(filter(lambda x: x % 2 == 1, data))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

En Python es posible «simular» un `filter()` a través de una *lista por comprensión*:

```
>>> [x for x in data if x % 2 == 1]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

### reduce()

Para poder usar esta función debemos usar el módulo `functools`. Nos permite aplicar una función dada sobre todos los elementos de un iterable de manera acumulativa. O dicho en otras palabras, nos permite **reducir** una función sobre un conjunto de valores. Supongamos que queremos realizar el producto de una serie de valores aplicando este enfoque:

```
>>> from functools import reduce

>>> def mult_values(a, b):
...     return a * b
...

>>> data = range(1, 6)

>>> reduce(mult_values, data)  # ((((1 * 2) * 3) * 4) * 5)
120
```

Aplicando una *función anónima «lambda»*…

```
>>> reduce(lambda x, y: x * y, data)
120
```

**Consejo:** Por cuestiones de legibilidad del código, se suelen preferir las **listas por comprensión** a funciones como `map()` o `filter()`, aunque cada problema tiene sus propias características y sus soluciones más adecuadas. Es un **enfoque «más pitónico»**.

## Hazlo pitónico

Trey Hunner explica en una de sus «newsletters» lo que él entiende por **código pitónico**:

«Pitónico es un término extraño que significa diferentes cosas para diferentes personas. Algunas personas piensan que código pitónico va sobre legibilidad. Otras personas piensan que va sobre adoptar características particulares de Python. Mucha gente tiene una definición difusa que no va sobre legibilidad ni sobre características del lenguaje.

Yo normalmente uso el término código pitónico como un sinónimo de código idiomático o la forma en la que la comunidad de Python tiende a hacer las cosas cuando escribe Python.

Eso deja mucho espacio a la interpretación, ya que lo que hace algo idiomático en Python no está particularmente bien definido.

Yo argumento que código pitónico implica adoptar el *desempaquetado de tuplas*, usar *listas por comprensión* cuando sea apropiado, usar *argumentos nominales* cuando tenga sentido, evitar el *uso excesivo de clases*, usar las *estructuras de iteración* adecuadas o evitar *recorrer mediante índices*.

Para mí, código pitónico significa intentar ver el código desde la perspectiva de las herramientas específicas que Python nos proporciona, en oposición a la forma en la que resolveríamos el mismo problema usando las herramientas que nos proporciona JavaScript, Java, C, …»

## Generadores

Un **generador**, como su propio nombre indica, se encarga de generar «valores» sobre los que podemos iterar. Es decir, no construye una secuencia de forma explícita, sino que nos permite ir «consumiendo» un valor de cada vez. Esta propiedad los hace idóneos para situaciones en las que el tamaño de las secuencias podría tener un impacto negativo en el consumo de memoria.

De hecho ya hemos visto algunos generadores y los hemos usado sin ser del todo conscientes. Algo muy parecido[8] a un generador es `range()` que ofrece la posibilidad de crear *secuencias de números*.

Básicamente existen dos implementaciones de generadores:

- Funciones generadoras.
- Expresiones generadoras.

**Importante:** A diferencia de las funciones ordinarias, los generadores tienen la capacidad de **«recordar» su estado** para recuperarlo en la siguiente iteración y continuar devolviendo nuevos valores.

### Funciones generadoras

Las funciones generadoras[9] (o **factorías de generadores**) se escriben como funciones ordinarias con el matiz de incorporar la sentencia `yield` que sustituye, de alguna manera, a `return`. Esta sentencia devuelve el valor indicado y, a la vez, «congela» el estado de la función hasta la siguiente llamada.

[8] La función `range()` es un tanto especial. Véase este artículo de Trey Hunner.

[9] Para una explicación detallada sobre generadores e iteradores se recomienda la ponencia Yield el amigo que no sabías que tenías de Jacobo de Vera.

Veamos un ejemplo en el que escribimos una **función generadora de números pares**:

```
>>> def evens(lim: int) -> int:
...     for i in range(0, lim + 1, 2):
...         yield i
...

>>> type(evens)
function

>>> evens_gen = evens(20)  # retorna un generador

>>> type(evens_gen)
generator
```

Una vez creado el generador, ya podemos iterar sobre él:

```
>>> for even in evens_gen:
...     print(even, end=' ')
...
0 2 4 6 8 10 12 14 16 18 20
```

De forma más «directa», podemos iterar sobre la propia llamada a la función generadora:

```
>>> for even in evens(20):
...     print(even, end=' ')
...
0 2 4 6 8 10 12 14 16 18 20
```

Si queremos «explicitar» la lista de valores que contiene un generador, podemos hacerlo convirtiendo a lista:

```
>>> list(evens(20))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Un detalle muy importante es que **los generadores «se agotan»**. Es decir, una vez que ya hemos consumido todos sus elementos, no obtendremos nuevos valores:

```
>>> evens_gen = evens(10)

>>> for even in evens_gen:
...     print(even, end=' ')
...
0 2 4 6 8 10

>>> for even in evens_gen:
...     print(even, end=' ')
```

(continué en la próxima página)

(proviene de la página anterior)

```
... # No sale nada... ¡Agotado!
```

### Expresiones generadoras

Una **expresión generadora** es sintácticamente muy similar a una *lista por comprensión*, pero utilizamos **paréntesis** en vez de corchetes. Se podría ver como una versión acortada de una función generadora.

Podemos tratar de reproducir el ejemplo visto en *funciones generadoras* en el que creamos números pares hasta el 20:

```
>>> evens_gen = (i for i in range(0, 20, 2))

>>> type(evens_gen)
generator

>>> for i in evens_gen:
...     print(i, end=' ')
...
0 2 4 6 8 10 12 14 16 18
```

**Ver también:**

Las expresiones generadoras admiten *condiciones* y *anidamiento de bucles*, tal y como se vio con las *listas por comprensión*.

Una expresión generadora se puede explicitar, sumar, buscar su máximo o su mínimo, o lo que queramos, tal y como lo haríamos con un iterable cualquiera:

```
>>> list(i for i in range(0, 20, 2))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> sum(i for i in range(0, 20, 2))
90

>>> min(i for i in range(0, 20, 2))
0

>>> max(i for i in range(0, 20, 2))
18
```

**Ejercicio**

pycheck: **gen_squared**

### Funciones interiores

Está permitido definir una función dentro de otra función:

```
>>> VALID_CHARS = 'xyz'

>>> def validation_rate(text: str) -> float:
...     '''Rate of valid chars in text.'''
...     def is_valid_char(char: str) -> bool:
...         return char in VALID_CHARS
...
...     checklist = [is_valid_char(c) for c in text]
...     return sum(checklist) / len(text)
...

>>> validation_rate('zxyzxxyz')
1.0

>>> validation_rate('abzxyabcdz')
0.4

>>> validation_rate('abc')
0.0
```

**Truco:** Estas funciones pueden tener sentido cuando su ámbito de aplicación es muy concreto y no se pueden reutilizar fácilmente.

### Clausuras

Una **clausura** (del término inglés «*closure*») establece el uso de una *función interior* que se genera dinámicamente y recuerda los valores de los argumentos con los que fue creada:

```
>>> def make_multiplier_of(n):
...     def multiplier(x):
...         return x * n
...     return multiplier
...

>>> m3 = make_multiplier_of(3)

>>> m5 = make_multiplier_of(5)

>>> type(m3)
```

(continué en la próxima página)

(proviene de la página anterior)

```
function

>>> m3(7)  # 7 * 3
21

>>> type(m5)
function

>>> m5(8)  # 8 * 5
40

>>> make_multiplier_of(5)(8)  # Llamada directa!
40
```

**Importante:** En una clausura retornamos una función, no una llamada a una función.

### Decoradores

Hay situaciones en las que necesitamos modificar el comportamiento de funciones existentes pero sin alterar su código. Para estos casos es muy útil usar decoradores.

Un **decorador** es una *función* que recibe como parámetro una función y devuelve otra función. Se podría ver como un caso particular de *clausura*.

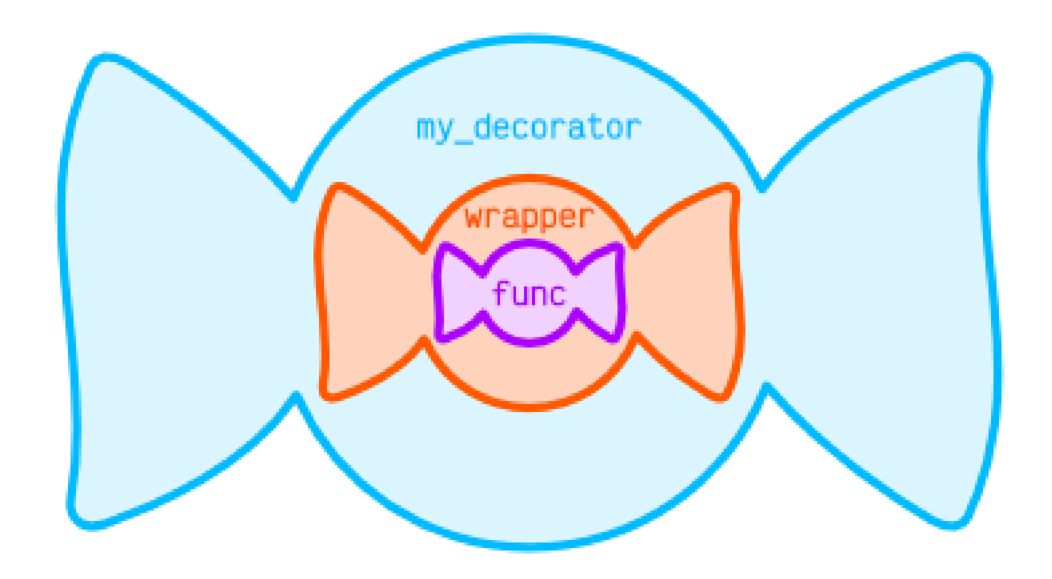



Figura 7: Esqueleto básico de un decorador

El *esqueleto básico* de un decorador es el siguiente:

```
>>> def my_decorator(func):
...     def wrapper(*args, **kwargs):
...         # some code before calling func
...         return func(*args, **kwargs)
...         # some code after calling func
...     return wrapper
...
```

| Elemento | Descripción |
|---|---|
| `my_decorator` | Nombre del decorador |
| `wrapper` | Función interior (convención de nombre) |
| `func` | Función a decorar (convención de nombre) |
| `*args` | Argumentos posicionales (convención de nombre) |
| `**kwargs` | Argumentos nominales (convención de nombre) |

Veamos un ejemplo de **decorador que convierte el resultado numérico de una función a su representación binaria**:

```
>>> def res2bin(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return bin(result)
...     return wrapper
...
```

Ahora definimos una función ordinaria (que usaremos más adelante) y que computa $x^n$:

```
>>> def power(x: int, n: int) -> int:
...     return x ** n
...

>>> power(2, 3)
8
>>> power(4, 5)
1024
```

Ahora aplicaremos el decorador definido previamente `res2bin()` sobre la función ordinaria `power()`. Se dice que `res2bin()` es la **función decoradora** y que `power()` es la **función decorada**:

```
>>> decorated_power = res2bin(power)

>>> decorated_power(2, 3)  # 8
```

(continué en la próxima página)

(proviene de la página anterior)

```
'0b1000'
>>> decorated_power(4, 5)  # 1024
'0b10000000000'
```

### Usando @ para decorar

Python nos ofrece un «syntactic sugar» para simplificar la aplicación de los decoradores a través del operador @ justo antes de la definición de la función que queremos decorar:

```
>>> @res2bin
... def power(x: int, n: int) -> int:
...     return x ** n
...

>>> power(2, 3)
'0b1000'
>>> power(4, 5)
'0b10000000000'
```

**Ejercicio**

pycheck: **abs_decorator**

### Manipulando argumentos

Hemos visto un ejemplo de decorador que trabaja sobre el resultado de la función decorada, pero nada impide que trabajemos sobre los argumentos que se le pasa a la función decorada.

Supongamos un escenario en el que implementamos **funciones que trabajan con dos operandos** y queremos asegurarnos de que **esos operados son números enteros**. Lo primero será definir el decorador:

```
>>> def assert_int(func):
...     def wrapper(value1: int, value2: int, /) -> int | float | None:
...         if isinstance(value1, int) and isinstance(value2, int):
...             return func(value1, value2)
...         return None
...     return wrapper
...
```

**Truco:** Dado que sabemos positivamente que las funciones a decorar trabajan con dos

operados (dos parámetros) podemos definir la función interior `wrapper(value1, value2)` con dos parámetros, en vez de con un número indeterminado de parámetros.

---

Ahora creamos una función sencilla que suma dos números y le aplicamos el decorador:

```
>>> @assert_int
... def _sum(a, b):
...     return a + b
...
```

Veamos el comportamiento para diferentes casos de uso:

```
>>> result = _sum(3, 4)
>>> print(result)
7

>>> result = _sum(5, 'a')
>>> print(result)
None

>>> result = _sum('a', 'b')
>>> print(result)
None
```

### Múltiples decoradores

Podemos aplicar más de un decorador a cada función. Para ejemplificarlo vamos a crear dos decoradores muy sencillos:

```
>>> def plus5(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result + 5
...     return wrapper
...

>>> def div2(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result // 2
...     return wrapper
...
```

Ahora aplicaremos ambos decoradores sobre una función que realiza el producto de dos números:

```
>>> @plus5
... @div2
... def prod(a, b):
...     return a * b
...

>>> prod(4, 3)
11

>>> ((4 * 3) // 2) + 5
11
```

Cuando tenemos varios decoradores **se aplican desde dentro hacia fuera** ya que la ejecución de un decorador depende de otro decorador.

Si anotamos los decoradores podemos ver exactamente cuál es el orden de ejecución:

```
>>> def plus5(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)  # ────────□
...         print(f'{result=}')             #         |
...         print('plus5')                  #         |
...         return result + 5               #         |
...     return wrapper                      #         |
...                                         #         |
...                                         #         |
... def div2(func):                         #         |
...     def wrapper(*args, **kwargs):       #         |
...         result = func(*args, **kwargs)  # □───────□
...         print(f'{result=}')
...         print('div2')
...         return result // 2
...     return wrapper
```

Ahora ejecutamos la función decorada:

```
>>> prod(4, 3)
result=12     # función prod "tal cual" (4*3)
div2          # decorador div2
result=6      # aplicación decorador div2 (12/2)
plus5         # decorador plus5
11            # aplicación decorador plus5 (6+5)
```

Una forma sencilla de entender el orden de ejecución de múltiples decoradores es aplicar las funciones decoradoras directamente sobre la función decorada.

Esto:

```
>>> @plus5
... @div2
... def prod(a, b):
...     return a * b
...
```

equivale a:

```
>>> plus5(div2(prod(4, 3)))
```

## Decoradores con parámetros

El último «salto mortal» sería definir decoradores con parámetros. El *esqueleto básico* de un decorador con parámetros es el siguiente:

```
>>> def my_decorator_with_params(*args, **kwargs):
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             return func(*args, **kwargs)
...         return wrapper
...     return decorator
...
```

**Atención:** Nótese que `my_decorator_with_params()` no es exactamente un decorador sino que es una factoría de decoradores (*clausura*) que devuelve un decorador según los argumentos pasados.

Lo más sencillo es verlo con un ejemplo. Supongamos que queremos forzar a que los parámetros de entrada a la función sean de un tipo concreto (pero parametrizable). Podríamos definir el decorador de la siguiente manera:

```
>>> def assert_type(atype):
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             all_args_with_atype = all(isinstance(a, atype) for a in args)
...             all_kwargs_with_atype = all(isinstance(a, atype) for a in kwargs.
↪values())
...             if all_args_with_atype and all_kwargs_with_atype:
...                 return func(*args, **kwargs)
...             return None
...         return wrapper
...     return decorator
...
```

Ahora creamos una función sencilla que suma dos números y le aplicamos el decorador:

```
>>> @assert_type(float)
... def _sum(a, b):
...     return a + b
...
```

Veamos el comportamiento para diferentes casos de uso:

```
>>> result = _sum(3, 4)
>>> print(result)
None

>>> result = _sum(3.0, 4.0)
>>> print(result)
7.0

>>> result = _sum(a=3.0, b=4.0)  # Funciona con kwargs!
>>> print(result)
7.0
```

La ventaja que tiene este enfoque es que podemos aplicar «distintos» decoradores modificando sus parámetros. Por ejemplo, supongamos que ahora queremos **asegurar que una función trabaja únicamente con cadenas de texto**:

```
>>> @assert_type(str)
... def split(text):
...     half_size = len(text) // 2
...     return text[:half_size], text[half_size:]
...
```

Veamos su aplicación con distintos tipos de datos:

```
>>> result = split('bienvenida')
>>> print(result)
('bienv', 'enida')

>>> result = split(256)
>>> print(result)
None

>>> result = split([10, 20, 30, 40])
>>> print(result)
None
```

**Ejercicio**

¿Sabría implementar un decorador para ordenar el resultado de cualquier función tomando un parámetro opcional que indique si la ordenación es ascendente o descendente?

---

### Funciones recursivas

La **recursividad** es el mecanismo por el cual una función se llama a sí misma:

```
>>> def call_me():
...     return call_me()
...

>>> call_me()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in call_me
  File "<stdin>", line 2, in call_me
  File "<stdin>", line 2, in call_me
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

> **Advertencia:** Podemos observar que existe un número máximo de llamadas recursivas. Python controla esta situación por nosotros, ya que, de no ser así, podríamos llegar a consumir todos los recursos del sistema.

Veamos ahora un ejemplo más real en el que computar el enésimo término de la Sucesión de Fibonacci utilizando una función recursiva:

```
>>> def fibonacci(n: int) -> int:
...     if n == 0:
...         return 0
...     if n == 1:
...         return 1
...     return fibonacci(n - 1) + fibonacci(n - 2)
...

>>> fibonacci(10)
55

>>> fibonacci(20)
6765
```

---

**Ejercicio**

pycheck: **factorial_recursive**

Otra aproximación a la recursividad se da en problemas donde tenemos que procesar una secuencia de elementos. Supongamos que nos piden **calcular la suma de las longitudes de una serie de palabras** definidas en una lista:

```
>>> def get_size(words: list[str]) -> int:
...     if len(words) == 0:
...         return 0
...     return len(words[0]) + get_size(words[1:])
...

>>> words = ['this', 'is', 'recursive']
>>> get_size(words)
15
```

### Funcionitis

La «funcionitis» es una «inflamación en la zona funcional» por querer aplicar funciones donde no es necesario. Un ejemplo vale más que mil explicaciones:

```
>>> def in_list(item: int, items: list[int]) -> bool:
...     return item in items
...

>>> in_list(1, [1, 2, 3])
True

>>> 1 in [1, 2, 3]  # That easy!
True
```

**Truco:** La «funcionitis» es uno de los síntomas de la llamada «sobre-ingeniería» a la que tendemos muchas de las personas que hacemos programación. Hay que intentar evitarla en la medida de lo posible.

### 6.1.5 Espacios de nombres

Como bien indica el *Zen de Python*:

> *Namespaces are one honking great idea – let's do more of those!*

Que vendría a traducirse como: «Los espacios de nombres son una gran idea – hagamos más de eso». Los **espacios de nombres** permiten definir **ámbitos** o **contextos** en los que agrupar nombres de objetos.

Los espacios de nombres proporcionan un mecanismo de empaquetado, de tal forma que podamos tener incluso nombres iguales que no hacen referencia al mismo objeto (siempre y cuando estén en ámbitos distintos).

Cada *función* define su propio espacio de nombres y es diferente del espacio de nombres global aplicable a todo nuestro programa.

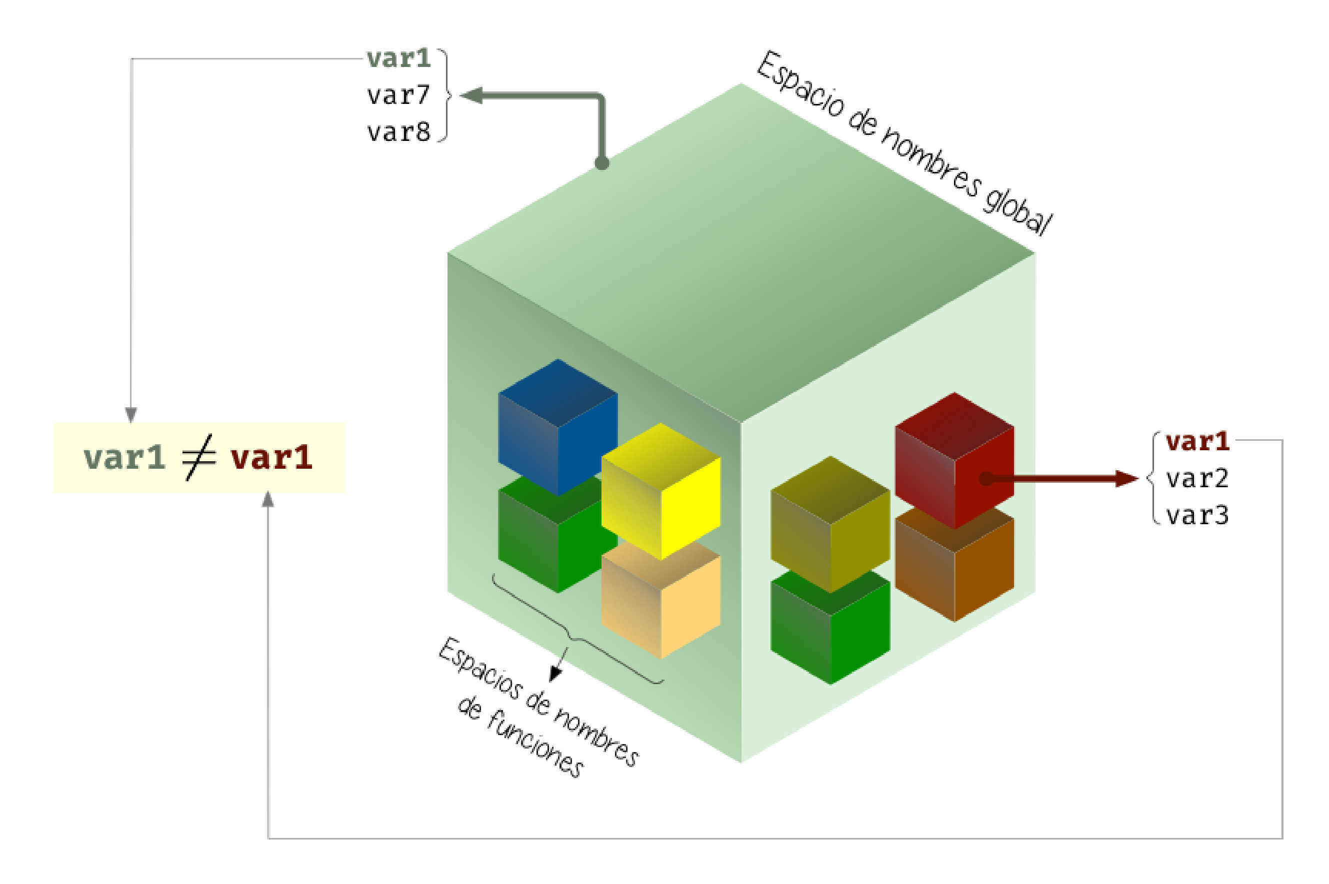



Figura 8: Espacio de nombres global vs espacios de nombres de funciones

### Acceso a variables globales

Cuando una variable se define en el *espacio de nombres global* podemos hacer uso de ella con total transparencia dentro del ámbito de las funciones del programa:

```
>>> language = 'castellano'

>>> def catalonia():
...     print(f'{language=}')
...

>>> language
'castellano'

>>> catalonia()
language='castellano'
```

### Creando variables locales

En el caso de que asignemos un valor a una variable global dentro de una función, no estaremos modificando ese valor. Por el contrario, estaremos creando una *variable en el espacio de nombres local*:

```
>>> language = 'castellano'

>>> def catalonia():
...     language = 'catalan'
...     print(f'{language=}')
...

>>> language
'castellano'

>>> catalonia()
language='catalan'

>>> language
'castellano'
```

### Forzando modificación global

Python nos permite modificar una variable definida en un espacio de nombres global dentro de una función. Para ello debemos usar el modificador `global`:

```
>>> language = 'castellano'

>>> def catalonia():
...     global language
...     language  = 'catalan'
...     print(f'{language=}')
...

>>> language
'castellano'

>>> catalonia()
language='catalan'

>>> language
'catalan'
```

> **Advertencia:** El uso de `global` no se considera una buena práctica ya que puede inducir a confusión y tener efectos colaterales indeseados.

### Contenido de los espacios de nombres

Python proporciona dos funciones para acceder al contenido de los espacios de nombres:

**`locals()`** Devuelve un diccionario con los contenidos del **espacio de nombres local**:

```
>>> language = 'castellano'

>>> def catalonia():
...     language  = 'catalan'
...     print(f'{locals()=}')
...

>>> catalonia()
locals()={'language': 'catalan'}
```

**`globals()`** Devuelve un diccionario con los contenidos del **espacio de nombres global**:

```
>>> globals()
{'__name__': '__main__',
'__doc__': 'Automatically created module for IPython interactive environment',
'__package__': None,
'__loader__': None,
'__spec__': None,
'__builtin__': <module 'builtins' (built-in)>,
'__builtins__': <module 'builtins' (built-in)>,
'_ih': ['',
"language = 'castellano'",
"def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\n    ",
'language',
'catalonia()',
'globals()'],
'_oh': {3: 'castellano'},
'_dh': ['/Users/sdelquin'],
'In': ['',
"language = 'castellano'",
"def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\n    ",
'language',
'catalonia()',
'globals()'],
'Out': {3: 'castellano'},
'get_ipython': <bound method InteractiveShell.get_ipython of <IPython.terminal.
↪interactiveshell.TerminalInteractiveShell object at 0x10e70c2e0>>,
'exit': <IPython.core.autocall.ExitAutocall at 0x10e761070>,
'quit': <IPython.core.autocall.ExitAutocall at 0x10e761070>,
'_': 'castellano',
'__': '',
'___': '',
'Prompts': IPython.terminal.prompts.Prompts,
'Token': Token,
'MyPrompt': __main__.MyPrompt,
'ip': <IPython.terminal.interactiveshell.TerminalInteractiveShell at␣
↪0x10e70c2e0>,
'_i': 'catalonia()',
'_ii': 'language',
'_iii': "def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\
↪n    ",
'_i1': "language = 'castellano'",
'language': 'castellano',
'_i2': "def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\
↪n    ",
'catalonia': <function __main__.catalonia()>,
'_i3': 'language',
'_3': 'castellano',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'_i4': 'catalonia()',
'_i5': 'globals()'}
```

### 6.1.6 Consejos para programar

**Chris Staudinger** comparte estos 7 consejos para mejorar tu código:

1. **Las funciones deberían hacer una única cosa.** *Por ejemplo, un mal diseño sería tener una única función que calcule el total de una cesta de la compra, los impuestos y los gastos de envío. Sin embargo esto se debería hacer con tres funciones separadas. Así conseguimos que el código sea más fácil de matener, reutilizar y depurar.*
2. **Utiliza nombres descriptivos y con significado.** *Los nombres autoexplicativos de variables y funciones mejoran la legibilidad del código. Por ejemplo – deberíamos llamar «total_cost» a una variable que se usa para almacenar el total de un carrito de la compra en vez de «x» ya que claramente explica su propósito.*
3. **No uses variables globales.** *Las variables globales pueden introducir muchos problemas, incluyendo efectos colaterales inesperados y errores de programación difíciles de trazar. Supongamos que tenemos dos funciones que comparten una variable global. Si una función cambia su valor la otra función podría no funcionar como se espera.*
4. **Refactorizar regularmente.** *El código inevitablemente cambia con el tiempo, lo que puede derivar en partes obsoletas, redundantes o desorganizadas. Trata de mantener la calidad del código revisando y refactorizando aquellas zonas que se editan.*
5. **No utilices «números mágicos» o valores «hard-codeados».** *No es lo mismo escribir «99 * 3» que «price * quantity». Esto último es más fácil de entender y usa variables con nombres descriptivos haciéndolo autoexplicativo. Trata de usar constantes o variables en vez de valores «hard-codeados».*
6. **Escribe lo que necesites ahora, no lo que pienses que podrías necesitar en el futuro.** *Los programas simples y centrados en el problema son más flexibles y menos complejos.*
7. **Usa comentarios para explicar el «por qué» y no el «qué».** *El código limpio es autoexplicativo y por lo tanto los comentarios no deberían usarse para explicar lo que hace el código. En cambio, los comentarios debería usarse para proporcionar contexto adicional, como por qué el código está diseñado de una cierta manera.*

### EJERCICIOS DE REPASO

1. pycheck: **num_in_interval**
2. pycheck: **extract_evens**
3. pycheck: **split_case**
4. pycheck: **perfect**
5. pycheck: **palindrome**
6. pycheck: **count_vowels**
7. pycheck: **pangram**
8. pycheck: **cycle_alphabet**
9. pycheck: **bubble_sort**
10. pycheck: **consecutive_seq**
11. pycheck: **magic_square**
12. pycheck: **sum_nested**
13. pycheck: **power_recursive**
14. pycheck: **hyperfactorial**

### AMPLIAR CONOCIMIENTOS

- Comparing Python Objects the Right Way: «is» vs «==»
- Python Scope & the LEGB Rule: Resolving Names in Your Code
- Defining Your Own Python Function
- Null in Python: Understanding Python's NoneType Object
- Python "!=" Is Not "is not": Comparing Objects in Python
- Python args and kwargs: Demystified
- Documenting Python Code: A Complete Guide
- Thinking Recursively in Python
- How to Use Generators and yield in Python
- How to Use Python Lambda Functions
- Python Decorators 101
- Writing Comments in Python

- Introduction to Python Exceptions
- Primer on Python Decorators

## 6.2 Objetos y Clases



Hasta ahora hemos estado usando objetos de forma totalmente transparente, casi sin ser conscientes de ello. Pero, en realidad, **todo en Python es un objeto**, desde números a funciones. El lenguaje provee ciertos mecanismos para no tener que usar explícitamente técnicas de orientación a objetos.

Llegados a este punto, investigaremos en profundidad la creación y manipulación de clases y objetos, así como todas las técnicas y procedimientos que engloban este paradigma.[1]

### 6.2.1 Programación orientada a objetos

La programación orientada a objetos (POO) o en sus siglas inglesas **OOP** es una manera de programar (paradigma) que permite llevar al código mecanismos usados con entidades de la vida real.

Sus **beneficios** son los siguientes:

[1] Foto original por Rabie Madaci en Unsplash.

**Encapsulamiento** Permite **empaquetar** el código dentro de una unidad (objeto) donde se puede determinar el ámbito de actuación.

**Abstracción** Permite **generalizar** los tipos de objetos a través de las clases y simplificar el programa.

**Herencia** Permite **reutilizar** código al poder heredar atributos y comportamientos de una clase a otra.

**Polimorfismo** Permite **crear** múltiples objetos a partir de una misma pieza flexible de código.

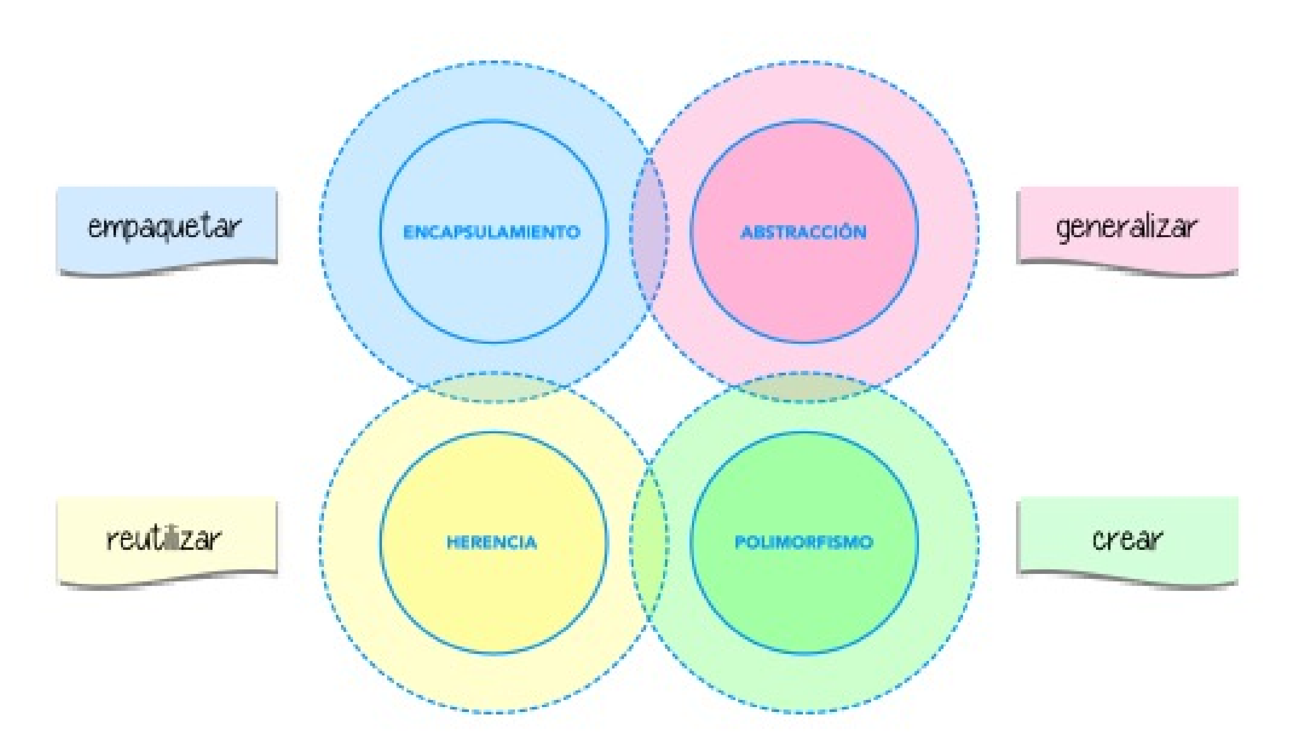



Figura 9: Beneficios de la Programación Orientada a Objetos

## ¿Qué es un objeto?

Un **objeto** es una **estructura de datos personalizada** que contiene **datos** y **código**:

| Elementos | ¿Qué son? | ¿Cómo se llaman? | ¿Cómo se identifican? |
|---|---|---|---|
| Datos | Variables | Atributos | Mediante sustantivos |
| Código | Funciones | Métodos | Mediante verbos |

Un objeto representa **una instancia única** de alguna entidad (a través de los valores de sus atributos) e interactúa con otros objetos (o consigo mismo) a través de sus métodos.

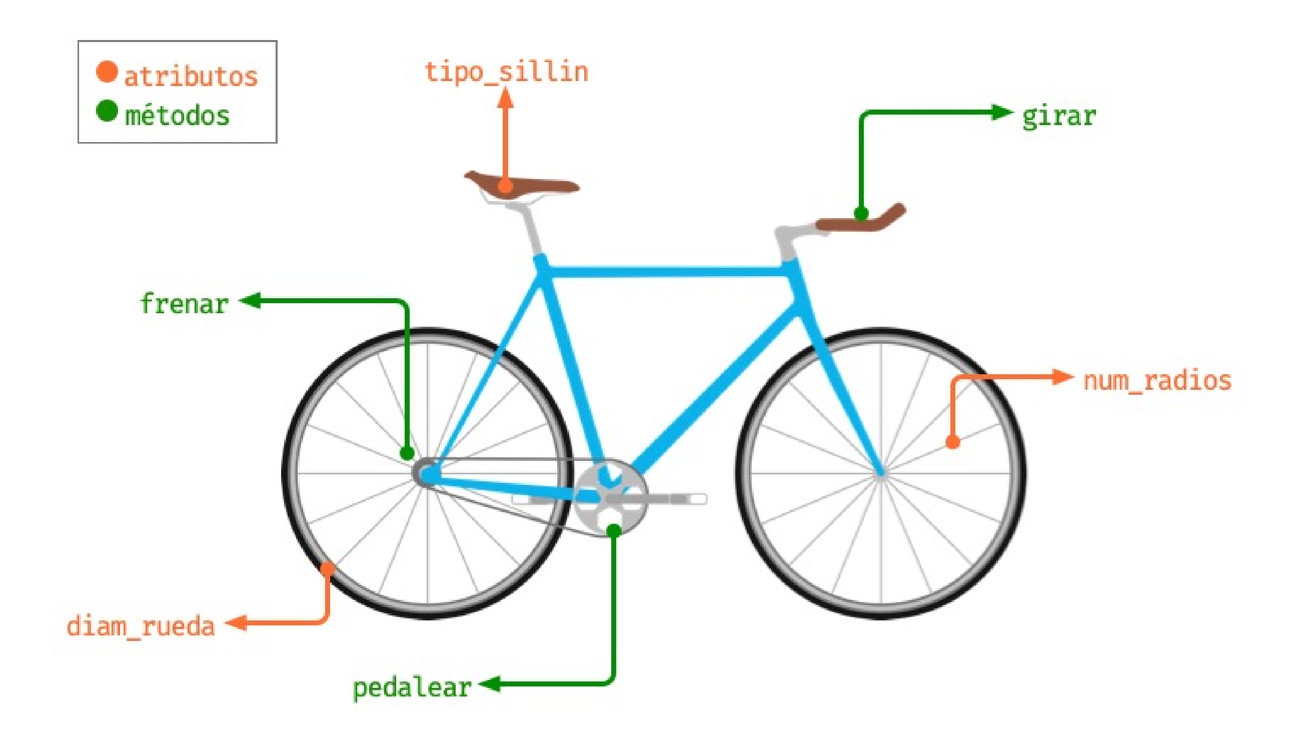



Figura 10: Analogía de atributos y métodos en un objeto «bicicleta»

### ¿Qué es una clase?

Para crear un objeto primero debemos definir la clase que lo contiene. Podemos pensar en la **clase** como el **molde** con el que se crean nuevos objetos de ese tipo.

En el **proceso de diseño** de una clase hay que tener en cuenta – entre otros – el **principio de responsabilidad única**[7], intentando que los atributos y los métodos que contenga esa clase estén enfocados a un *objetivo único y bien definido*.

## 6.2.2 Creando objetos

Empecemos por crear nuestra **primera clase**. En este caso vamos a modelar algunos de los droides de la saga StarWars:

Para ello usaremos la palabra reservada `class` seguida del nombre de la clase:

```
>>> class StarWarsDroid:
...     pass
...
```

[7] Principios SOLID
[2] Fuente de la imagen: Astro Mech Droids.

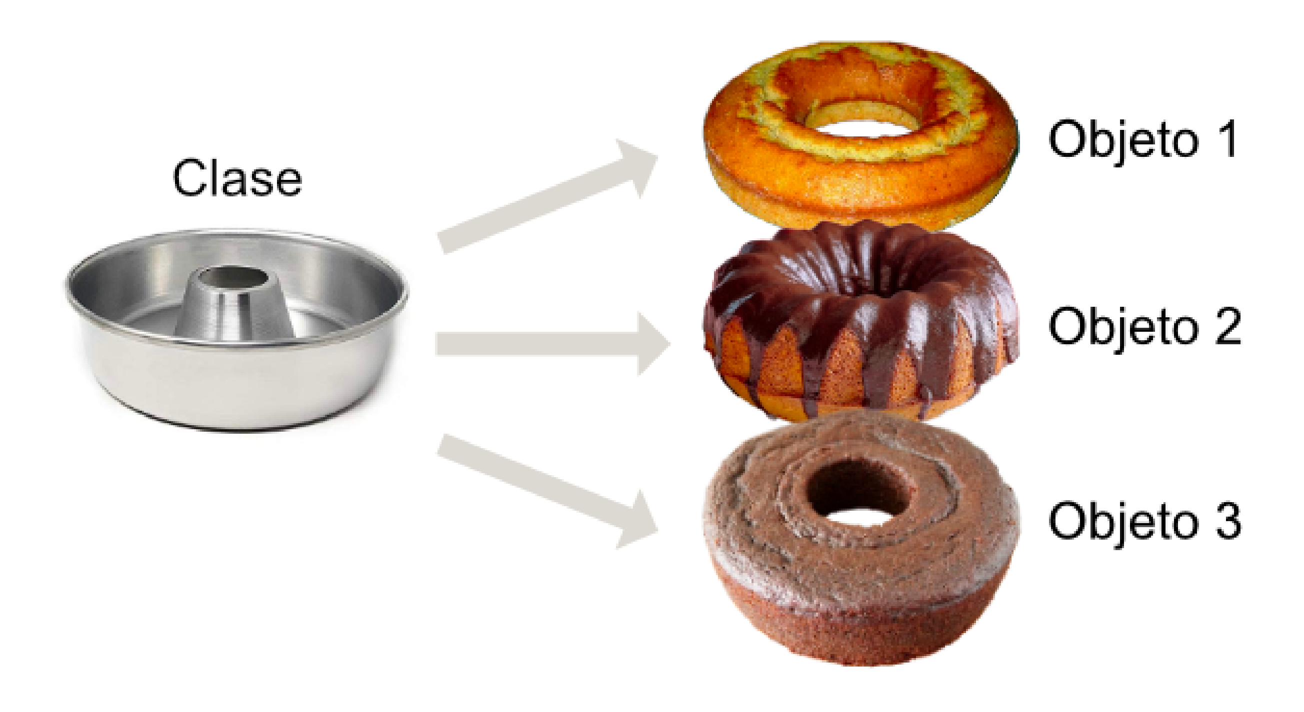



Figura 11: Ejemplificación de creación de objetos a partir de una clase



Figura 12: Droides de la saga StarWars[2]

**Consejo:** Los nombres de clases se suelen escribir en formato `CamelCase` y en singular[3].

Existen multitud de droides en el universo StarWars. Una vez que hemos definido la clase genérica podemos crear **instancias/objetos** (droides) concretos:

```
>>> c3po = StarWarsDroid()
>>> r2d2 = StarWarsDroid()
>>> bb8 = StarWarsDroid()

>>> type(c3po)
__main__.StarWarsDroid
>>> type(r2d2)
__main__.StarWarsDroid
>>> type(bb8)
__main__.StarWarsDroid
```

### Añadiendo métodos

Un **método** es una función que forma parte de una clase o de un objeto. En su ámbito tiene acceso a otros métodos y atributos de la clase o del objeto al que pertenece.

La definición de un método (de instancia) es análoga a la de una función ordinaria, pero incorporando un primer parámetro `self` que hace referencia a la instancia actual del objeto.

Una de las acciones más sencillas que se pueden hacer sobre un droide es encenderlo o apagarlo. Vamos a implementar estos dos métodos en nuestra clase:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...

>>> k2so = Droid()

>>> k2so.switch_on()
Hi! I'm a droid. Can I help you?

>>> k2so.switch_off()
Bye! I'm going to sleep
```

[3] Guía de estilos PEP8 para convenciones de nombres.

**Consejo:** El nombre `self` es sólo una convención. Este parámetro puede llamarse de otra manera, pero seguir el estándar ayuda a la legibilidad.

### Añadiendo atributos

Un **atributo** no es más que una variable, un nombre al que asignamos un valor, con la particularidad de vivir dentro de una clase o de un objeto.

Supongamos que, siguiendo con el ejemplo anterior, queremos guardar en un atributo el estado del droide (encendido/apagado):

```
>>> class Droid:
...     def switch_on(self):
...         self.power_on = True
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         self.power_on = False
...         print("Bye! I'm going to sleep")

>>> k2so = Droid()

>>> k2so.switch_on()
Hi! I'm a droid. Can I help you?
>>> k2so.power_on
True

>>> k2so.switch_off()
Bye! I'm going to sleep
>>> k2so.power_on
False
```

**Importante:** Siempre que queramos acceder a cualquier método o atributo del objeto habrá que utilizar la palabra `self`.

### Inicialización

Existe un **método especial** que se ejecuta cuando creamos una instancia de un objeto. Este método es `__init__` y nos permite asignar atributos y realizar operaciones con el objeto en el momento de su creación. También es ampliamente conocido como el **constructor**.

Veamos un ejemplo de este método con nuestros droides en el que únicamente guardaremos el nombre del droide como un atributo del objeto:

```
1 >>> class Droid:
2 ...     def __init__(self, name: str):
3 ...         self.name = name
4 ...
5
6 >>> droid = Droid('BB-8')
7
8 >>> droid.name
9 'BB-8'
```

**Línea 2** Definición del constructor.

**Línea 7** Creación del objeto (y llamada implícita al constructor)

**Línea 9** Acceso al atributo `name` creado previamente en el constructor.

Es importante tener en cuenta que si no usamos `self` estaremos creando una variable local en vez de un atributo del objeto:

```
>>> class Droid:
...     def __init__(self, name: str):
...         name = name  # No lo hagas!
...

>>> droid = Droid('BB-8')

>>> droid.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute 'name'
```

**Ejercicio**

Escriba una clase `MobilePhone` que represente un teléfono móvil.

Atributos:

- `manufacturer` (cadena de texto)
- `screen_size` (flotante)

- `num_cores` (entero)
- `apps` (lista de cadenas de texto)
- `status` (`False`: apagado, `True`: encendido)

Métodos:

- `__init__(self, manufacturer, screen_size, num_cores)`
- `power_on(self)`
- `power_off(self)`
- `install_app(self, app)` *(no instalar la app si ya existe)*
- `uninstall_app(self, app)` *(no borrar la app si no existe)*

*¿Serías capaz de extender el método* `install_app()` *para instalar varias aplicaciones a la vez?*

---

### 6.2.3 Atributos

#### Acceso directo

En el siguiente ejemplo vemos que, aunque el atributo `name` se ha creado en el constructor de la clase, también podemos modificarlo desde «fuera» con un acceso directo:

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.name = name
...

>>> droid = Droid('C-3PO')

>>> droid.name
'C-3PO'

>>> droid.name = 'waka-waka'  # esto sería válido!
```

Python nos permite **añadir atributos dinámicamente** a un objeto incluso después de su creación:

```
>>> droid.manufacturer = 'Cybot Galactica'
>>> droid.height = 1.77
```

**Nota:** Nótese el acceso a los atributos con `obj.attribute` en vez de lo que veníamos usando en *diccionarios* donde hay que escribir «un poco más» `obj['attribute']`.

### Propiedades

Como hemos visto previamente, los atributos definidos en un objeto son accesibles públicamente. Esto puede parecer extraño a personas que vengan de otros lenguajes de programación (véase Java). En Python existe un cierto «sentido de la responsabilidad» a la hora de programar y manejar este tipo de situaciones: Casi todo es posible a priori pero se debe controlar explícitamente.

Una primera solución «pitónica» para la privacidad de los atributos es el uso de **propiedades**. La forma más común de aplicar propiedades es mediante el uso de *decoradores*:

- `@property` para leer el valor de un atributo («getter»).
- `@name.setter` para escribir el valor de un atributo.

Veamos un ejemplo en el que estamos ofuscando el nombre del droide a través de propiedades:

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.hidden_name = name
...
...     @property
...     def name(self) -> str:
...         print('inside the getter')
...         return self.hidden_name
...
...     @name.setter
...     def name(self, name: str) -> None:
...         print('inside the setter')
...         self.hidden_name = name
...

>>> droid = Droid('N1-G3L')

>>> droid.name
inside the getter
'N1-G3L'

>>> droid.name = 'Nigel'
inside the setter

>>> droid.name
```

(continué en la próxima página)

(proviene de la página anterior)

```
inside the getter
'Nigel'
```

En cualquier caso, seguimos pudiendo acceder directamente a `.hidden_name`:

```
>>> droid.hidden_name
'Nigel'
```

Incluso podemos cambiar su valor:

```
>>> droid.hidden_name = 'waka-waka'

>>> droid.name
inside the getter
'waka-waka'
```

### Valores calculados

Una propiedad también se puede usar para devolver un **valor calculado** (o computado).

A modo de ejemplo, supongamos que la altura del periscopio de los droides astromecánicos se calcula siempre como un porcentaje de su altura. Veamos cómo implementarlo:

```
>>> class AstromechDroid:
...     def __init__(self, name: str, height: float):
...         self.name = name
...         self.height = height
...
...     @property
...     def periscope_height(self) -> float:
...         return 0.3 * self.height
...

>>> droid = AstromechDroid('R2-D2', 1.05)

>>> droid.periscope_height  # podemos acceder como atributo
0.315

>>> droid.periscope_height = 10  # no podemos modificarlo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Las propiedades **no pueden recibir parámetros** ya que no tiene sentido semánticamente:

```
>>> class AstromechDroid:
...     def __init__(self, name: str, height: float):
...         self.name = name
...         self.height = height
...
...     @property
...     def periscope_height(self, from_ground: bool = False) -> float:
...         height_factor = 1.3 if from_ground else 0.3
...         return height_factor * self.height
...

>>> droid = AstromechDroid('R2-D2', 1.05)

>>> droid.periscope_height
0.315

>>> droid.periscope_height(from_ground=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object is not callable
```

En este caso tendríamos que implementar un método para resolver el escenario planteado.

**Consejo:** La ventaja de usar valores calculados sobre simples atributos es que el cambio de valor en un atributo no asegura que actualicemos otro atributo, y además siempre podremos modificar directamente el valor del atributo, con lo que podríamos obtener efectos colaterales indeseados.

### Cacheando propiedades

En los ejemplos anteriores hemos creado una propiedad que calcula el alto del periscopio de un droide astromecánico a partir de su altura. El «coste» de este cálculo es bajo, pero imaginemos por un momento que fuera muy alto.

Si cada vez que accedemos a dicha propiedad tenemos que realizar ese cálculo, estaríamos siendo muy ineficientes (en el caso de que la altura del droide no cambiara). Veamos una aproximación a este escenario usando el **cacheado de propiedades**:

```
>>> class AstromechDroid:
...     def __init__(self, name: str, height: float):
...         self.name = name
...         self.height = height  # llamada al setter
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     @property
...     def height(self) -> float:
...         return self._height
...
...     @height.setter
...     def height(self, height: float) -> None:
...         self._height = height
...         self._periscope_height = None  # invalidar caché
...
...     @property
...     def periscope_height(self) -> float:
...         if self._periscope_height is None:
...             print('Calculating periscope height...')
...             self._periscope_height = 0.3 * self.height
...         return self._periscope_height
```

Probamos ahora la implementación diseñada, modificando la altura del droide:

```
>>> droid = AstromechDroid('R2-D2', 1.05)

>>> droid.periscope_height
Calculating periscope height...
0.315
>>> droid.periscope_height  # Cacheado!
0.315

>>> droid.height = 1.15

>>> droid.periscope_height
Calculating periscope height...
0.345
>>> droid.periscope_height  # Cacheado!
0.345
```

### Ocultando atributos

Python tiene una convención sobre aquellos atributos que queremos hacer **«privados»** (u ocultos): comenzar el nombre con doble subguión `__`

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.__name = name
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> droid = Droid('BC-44')

>>> droid.__name  # efectivamente no aparece como atributo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute '__name'
```

Lo que realmente ocurre tras el telón se conoce como «*name mangling*» y consiste en modificar el nombre del atributo incorporado la clase como un prefijo. Sabiendo esto podemos acceder al valor del atributo supuestamente privado:

```
>>> droid._Droid__name
'BC-44'
```

**Nota:** La filosofía de Python permite hacer casi cualquier cosa con los objetos que se manejan, eso sí, el sentido de la responsabilidad se traslada a la persona que desarrolla e incluso a la persona que hace uso del objeto.

### Atributos de clase

Podemos asignar atributos a una clase y serán asumidos por todos los objetos instanciados de esa clase.

A modo de ejemplo, en un principio, todos los droides están diseñados para que obedezcan a su dueño. Esto lo conseguiremos a nivel de clase, salvo que ese comportamiento se sobreescriba:

```
>>> class Droid:
...     obeys_owner = True  # obedece a su dueño
...

>>> good_droid = Droid()
>>> good_droid.obeys_owner
True

>>> t1000 = Droid()  # T-1000 (Terminator)
>>> t1000.obeys_owner = False
>>> t1000.obeys_owner
False

>>> Droid.obeys_owner  # el cambio no afecta a nivel de clase
True
```

**Truco:** Los atributos de clase son accesibles tanto desde la clase como desde las instancias creadas.

Hay que tener en cuenta lo siguiente:

- Si modificamos un atributo de clase desde un objeto, sólo modificamos el valor en el objeto y no en la clase.
- Si modificamos un atributo de clase desde una clase, **modificamos el valor en todos los objetos pasados y futuros**.

Veamos un ejemplo de esto último:

```
>>> class Droid:
...     obeys_owner = True
...

>>> droid1 = Droid()
>>> droid1.obeys_owner
True

>>> droid2 = Droid()
>>> droid2.obeys_owner
True

>>> Droid.obeys_owner = False  # cambia pasado y futuro

>>> droid1.obeys_owner
False
>>> droid2.obeys_owner
False

>>> droid3 = Droid()
>>> droid3.obeys_owner
False
```

## 6.2.4 Métodos

### Métodos de instancia

Un **método de instancia** es un método que modifica o accede al estado del objeto al que hace referencia. Recibe `self` como primer parámetro, el cual se convierte en el propio objeto sobre el que estamos trabajando. Python envía este argumento de forma transparente: no hay que pasarlo como argumento.

Veamos un ejemplo en el que, además del constructor, creamos un método de instancia para hacer que un droide se mueva:

```
>>> class Droid:
...     def __init__(self, name: str):  # método de instancia -> constructor
...         self.name = name
...         self.covered_distance = 0
...
...     def move_up(self, steps: int) -> None:  # método de instancia
...         self.covered_distance += steps
...         print(f'Moving {steps} steps')
...

>>> droid = Droid('C1-10P')

>>> droid.move_up(10)
Moving 10 steps
```

### Propiedades vs Métodos

Es razonable plantearse cuándo usar *propiedades* o cuándo usar *métodos de instancia*.

Si la implementación requiere de parámetros, no hay confusión, necesitamos usar métodos.

Pero más allá de esto, no existe una respuesta clara y concisa a la pregunta. Aunque sí podemos dar algunas «pistas» para saber cuándo usar propiedades o cuándo usar métodos:

### Métodos de clase

Un **método de clase** es un método que modifica o accede al estado de la clase a la que hace referencia. Recibe `cls` como primer parámetro, el cual se convierte en la propia clase sobre la que estamos trabajando. Python envía este argumento de forma transparente. La identificación de estos métodos se completa aplicando el decorador `@classmethod` a la función.

Veamos un ejemplo en el que implementamos un método de clase que **muestra el número de droides creados**:

```
>>> class Droid:
...     count = 0
...
...     def __init__(self):
...         Droid.count += 1
...
...     @classmethod
```

(continué en la próxima página)

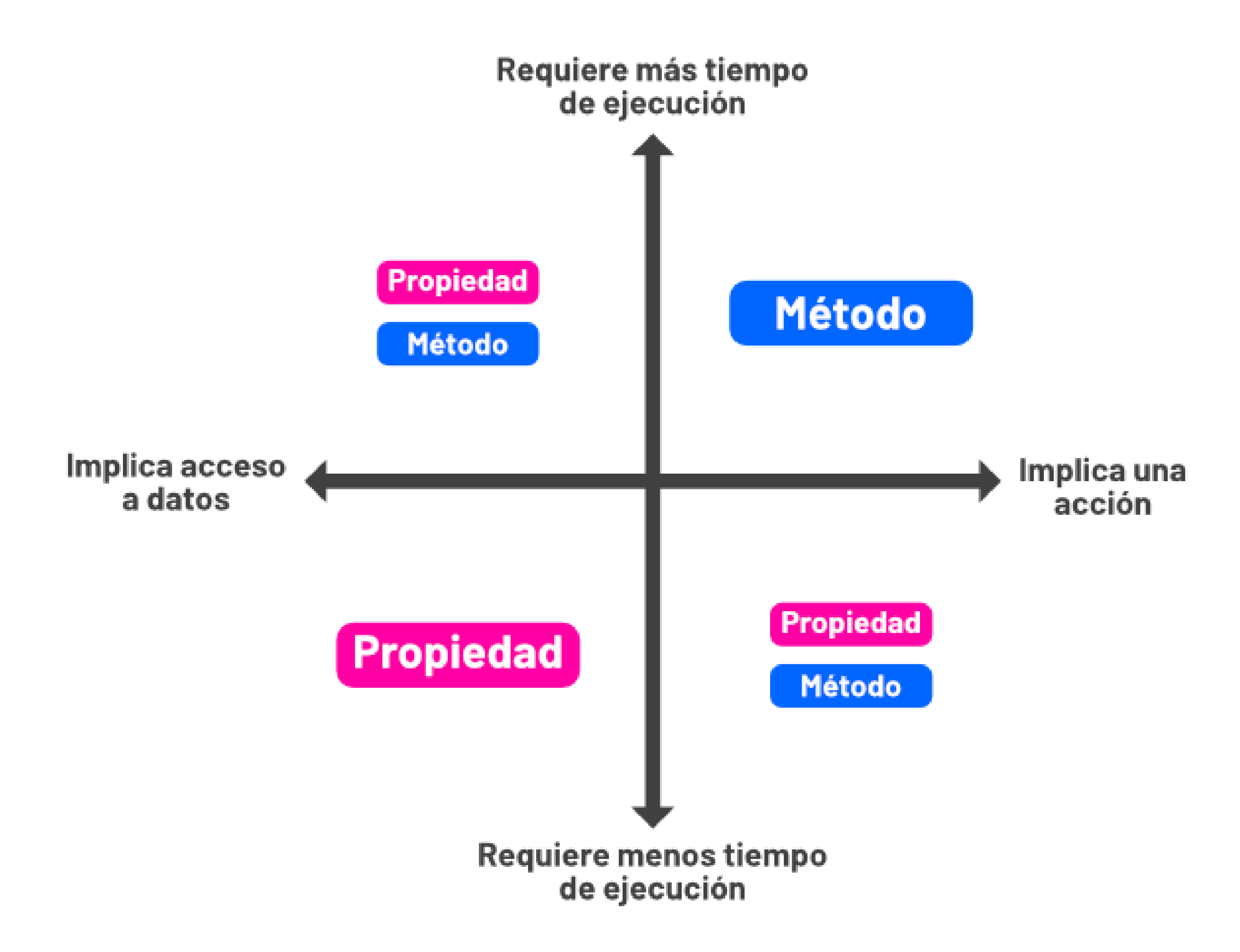



Figura 13: ¿Cuándo usar propiedades vs métodos?

(proviene de la página anterior)

```
...     def total_droids(cls) -> None:
...         print(f'{cls.count} droids built so far!')
...

>>> droid1 = Droid()
>>> droid2 = Droid()
>>> droid3 = Droid()

>>> Droid.total_droids()
3 droids built so far!
```

**Consejo:** El nombre `cls` es sólo una convención. Este parámetro puede llamarse de otra manera, pero seguir el estándar ayuda a la legibilidad.

### Métodos estáticos

Un **método estático** es un método que no «debería» modificar el estado del objeto ni de la clase. No recibe ningún parámetro especial. La identificación de estos métodos se completa aplicando el decorador `@staticmethod` a la función.

Veamos un ejemplo en el que creamos un método estático para devolver las categorías de droides que existen en StarWars:

```
>>> class Droid:
...     def __init__(self):
...         pass
...
...     @staticmethod
...     def get_droids_categories() -> tuple[str]:
...         return ('Messeger', 'Astromech', 'Power', 'Protocol')
...

>>> Droid.get_droids_categories()
('Messeger', 'Astromech', 'Power', 'Protocol')
```

### Métodos decorados

Es posible que, según el escenario, queramos **decorar ciertos métodos** de nuestra clase. Esto lo conseguiremos siguiendo la misma estructura de *decoradores* que ya hemos visto, pero con ciertos matices.

A continuación veremos un ejemplo en el que creamos un decorador para auditar las acciones de un droide y saber quién ha hecho qué:

```
>>> class Droid:
...     @staticmethod
...     def audit(method):
...         def wrapper(self, *args, **kwargs):
...             print(f'Droid {self.name} running {method.__name__}')
...             return method(self, *args, **kwargs)  # Ojo llamada!
...         return wrapper
...
...     def __init__(self, name: str):
...         self.name = name
...         self.pos = [0, 0]
...
...     @audit
...     def move(self, x: int, y: int):
...         self.pos[0] += x
...         self.pos[1] += y
...
...     @audit
...     def reset(self):
...         self.pos = [0, 0]

>>> droid = Droid('B1')

>>> droid.move(1, 1)
Droid B1 running move

>>> droid.reset()
Droid B1 running reset
```

A tener en cuenta la llamada al método de instancia dentro del decorador:

```
>>> method(self, *args, **kwargs) == self.method(*args, **kwargs)
```

El decorador se puede poner dentro o fuera de la clase. Por una cuestión de encapsulamiento podría tener sentido dejarlo **dentro de la clase como método estático**.

**Ver también:**

También es posible aplicar esta misma técnica usando *decoradores con parámetros*.

### Métodos mágicos

#### Nivel avanzado

Cuando escribimos `'hello world' * 3` ¿cómo sabe el objeto `'hello world'` lo que debe hacer para multiplicarse con el objeto entero `3`? O dicho de otra forma, ¿cuál es la implementación del operador `*` para «strings» e «int»? En valores numéricos puede parecer evidente (siguiendo los operadores matemáticos), pero no es así para otros objetos. La solución que proporciona Python para estas (y otras) situaciones son los **métodos mágicos**.

Los métodos mágicos empiezan y terminan por doble subguión `__` (es por ello que también se les conoce como «dunder-methods»). Uno de los «dunder-methods» más famosos es el constructor de una clase: `__init__()`.

---

**Importante:** Digamos que los métodos mágicos se «disparan» de manera transparente cuando utilizamos ciertas estructuras y expresiones del lenguaje.

---

Para el caso de los operadores, existe un método mágico asociado (que podemos personalizar). Por ejemplo la comparación de dos objetos se realiza con el método `__eq__()`:

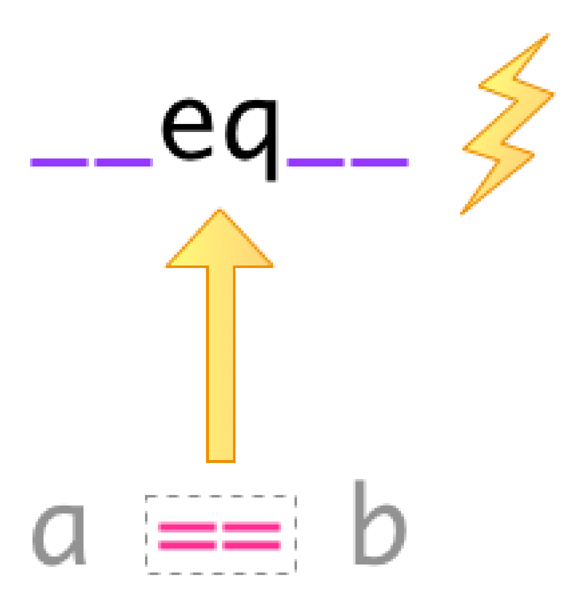



Figura 14: Equivalencia entre operador y método mágico

Extrapolando esta idea a nuestro universo StarWars, podríamos establecer que dos droides son iguales si su nombre es igual, independientemente de que tengan distintos números de serie:

```
>>> class Droid:
...     def __init__(self, name: str, serial_number: int):
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         self.name = name
...         self.serial_number = serial_number
...
...     def __eq__(self, droid: Droid) -> bool:
...         return self.name == droid.name
...

>>> droid1 = Droid('C-3PO', 43974973242)
>>> droid2 = Droid('C-3PO', 85094905984)

>>> droid1 == droid2  # llamada implícita a __eq__
True

>>> droid1.__eq__(droid2)
True
```

**Truco:**

Para poder utilizar la anotación de tipo `Droid` necesitamos añadir la siguiente línea al principio de nuestro código:

`from __future__ import annotations`

**Nota:** Los métodos mágicos no sólo están restringidos a operadores de comparación o matemáticos. Existen muchos otros en la documentación oficial de Python, donde son llamados métodos especiales.

Veamos un ejemplo en el que **«sumamos» dos droides** (*esto se podría ver como una fusión*). Supongamos que la suma de dos droides implica: a) que el nombre del droide resultante es la concatenación de los nombres de los droides de entrada; b) que la energía del droide resultante es la suma de la energía de los droides de entrada:

```
>>> class Droid:
...     def __init__(self, name: str, power: int):
...         self.name = name
...         self.power = power
...
...     def __add__(self, other: Droid) -> Droid:
...         new_name = self.name + '-' + other.name
...         new_power = self.power + other.power
...         return Droid(new_name, new_power)  # Hay que devolver un objeto de tipo␣
↪Droid
...
```

(continué en la próxima página)

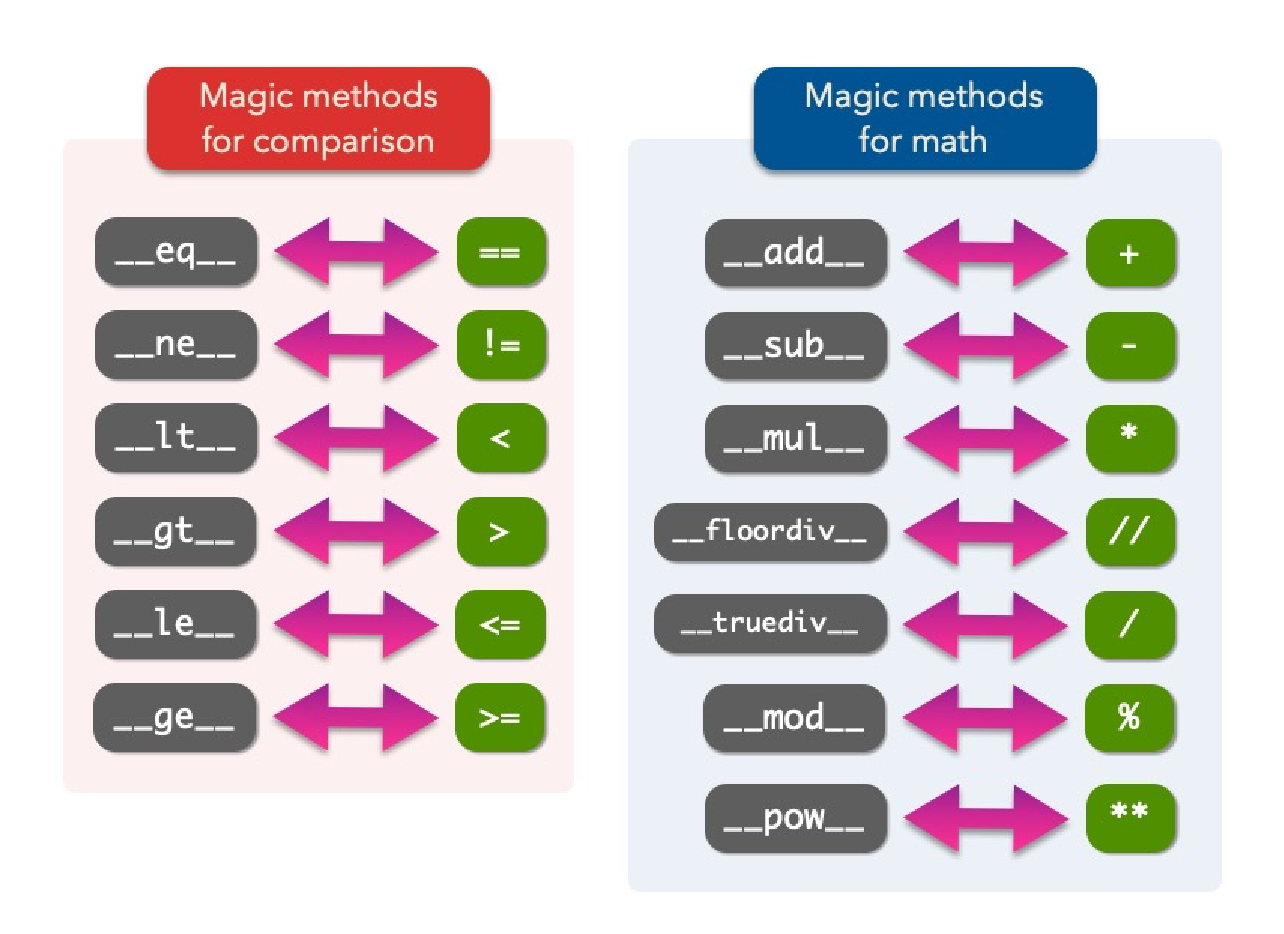



Figura 15: Métodos mágicos para comparaciones y operaciones matemáticas

(proviene de la página anterior)

```
>>> droid1 = Droid('C3PO', 45)
>>> droid2 = Droid('R2D2', 91)

>>> droid3 = droid1 + droid2

>>> print(f'Fusion droid:\n{droid3.name} with power {droid3.power}')
Fusion droid:
C3PO-R2D2 with power 136
```

**Importante:** Este tipo de operaciones debe **devolver un objeto** de la clase con la que estamos trabajando.

**Truco:** En este tipo de métodos mágicos el parámetro suele llamarse `other` haciendo referencia al «otro» objeto que entra en la operación. Es una convención.

### Sobrecarga de operadores

¿Qué ocurriría si sumamos un número entero a un droide? De primeras nada, porque no lo tenemos contemplado, pero podríamos establecer un significado: Si sumamos un número entero a un droide éste aumenta su energía en el valor indicado. Vamos a intentar añadir también este comportamiento al operador suma ya implementado.

Aunque en Python no existe técnicamente la «sobrecarga de funciones», sí que podemos simularla identificando el tipo del objeto que nos pasan y realizando acciones en base a ello:

```
>>> class Droid:
...     def __init__(self, name: str, power: int):
...         self.name = name
...         self.power = power
...
...     def __add__(self, other: Droid | int) -> Droid:
...         if isinstance(other, Droid):
...             new_name = self.name + '-' + other.name
...             new_power = self.power + other.power
...         elif isinstance(other, int):
...             new_name = self.name
...             new_power = self.power + other
...         return Droid(new_name, new_power)
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> droid = Droid('L3-37', 75)

>>> powerful_droid = droid + 25

>>> powerful_droid.power
100
```

Esta misma estrategia se puede aplicar al **operador de igualdad** ya que es muy habitual encontrar comparaciones de objetos en nuestro código. Por ello, deberíamos tener en cuenta si se van a comparar dos objetos de distinta naturaleza.

Retomando el caso ya visto… **¿qué pasaría si comparamos un droide con una cadena de texto?**

```
>>> class Droid:
...     def __init__(self, name: str, serial_number: int):
...         self.name = name
...         self.serial_number = serial_number
...
...     def __eq__(self, droid: Droid) -> bool:
...         return self.name == droid.name
...

>>> droid = Droid('C-3PO', 43974973242)

>>> droid == 'C-3PO'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in __eq__
AttributeError: 'str' object has no attribute 'name'
```

No funciona. Debemos contemplar el caso donde recibimos un objeto «cualquiera» a la hora de comparar. Veamos una posible implementación del operador de igualdad:

```
>>> class Droid:
...     def __init__(self, name: str, serial_number: int):
...         self.name = name
...         self.serial_number = serial_number
...
...     def __eq__(self, other: Droid | object) -> bool:
...         if isinstance(other, Droid):
...             return self.name == droid.name
...         return False
...
```

Ahora podemos comprobar que todo funciona como esperaríamos:

```
>>> droid = Droid('C-3PO', 43974973242)

>>> droid == 'C-3PO'
False
```

### __str__

Uno de los métodos mágicos más utilizados es `__str__` y permite establecer la forma en la que un objeto es representado como *cadena de texto*:

```
>>> class Droid:
...     def __init__(self, name: str, serial_number: int):
...         self.serial_number = serial_number
...         self.name = name
...
...     def __str__(self) -> str:
...         return f'🤖 Droid "{self.name}" serial-no {self.serial_number}'
...

>>> droid = Droid('K-2SO', 8403898409432)

>>> print(droid)  # llamada a droid.__str__()
🤖 Droid "K-2SO" serial-no 8403898409432

>>> str(droid)
'🤖 Droid "K-2SO" serial-no 8403898409432'

>>> f'Droid -> {droid}'
'Droid -> 🤖 Droid "K-2SO" serial-no 8403898409432'
```

**Ejercicio**

Defina una clase `Fraction` que represente una fracción con numerador y denominador enteros y utilice los métodos mágicos para poder sumar, restar, multiplicar y dividir estas fracciones.

Además de esto, necesitaremos:

- `gcd(a, b)` como **método estático** siguiendo el *algoritmo de Euclides* para calcular el máximo común divisor entre `a` y `b`.
- `__init__(self, num, den)` para construir una fracción (incluyendo simplificación de sus términos mediante el método `gcd()`.
- `__str__(self)` para representar una fracción.

Algoritmo de Euclides:

```
def gcd(a: int, b: int) -> int:
    ''' Algoritmo de Euclides para el cálculo del Máximo Común Divisor. '''
    while b > 0:
        a, b = b, a % b
    return a
```

Compruebe que se cumplen las siguientes igualdades:

$$\left[\frac{25}{30} + \frac{40}{45} = \frac{31}{18}\right] \quad \left[\frac{25}{30} - \frac{40}{45} = \frac{-1}{18}\right] \quad \left[\frac{25}{30} * \frac{40}{45} = \frac{20}{27}\right] \quad \left[\frac{25}{30} / \frac{40}{45} = \frac{15}{16}\right]$$

### __repr__

En ausencia del método `__str__()` se usará por defecto el método `__repr__()`. La diferencia entre ambos métodos es que el primero está más pensado para una representación del objeto de cara al usuario mientras que el segundo está más orientado al desarrollador.

El método `__repr()__` **se invoca automáticamente** en los dos siguientes escenarios:

1. Cuando no existe el método `__str__()` en el objeto y tratamos de encontrar su representación en cadena de texto con `str()` o `print()`.
2. Cuando utilizamos el intérprete interactivo de Python y pedimos el «valor» del objeto.

Veamos un ejemplo. En primer lugar un droide que sólo implementa el método `__str__()`:

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.name = name
...
...     def __str__(self):
...         return f"Hi there! I'm {self.name}"
...

>>> c14 = Droid('C-14')

>>> print(c14)  # __str()__
Hi there! I'm C-14

>>> c14  # __repr()__
<__main__.Droid at 0x103d7cc10>
```

Ahora implementamos también el método `__repr__()`:

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.name = name
...
...     def __str__(self):
...         return f"Hi there! I'm {self.name}"
...
...     def __repr__(self):
...         return f"[Droid] '{self.name}' @ {hex(id(self))}"
...

>>> c14 = Droid('C-14')

>>> print(c14)
Hi there! I'm C-14

>>> c14  # __repr__()
[Droid] 'C-14' @ 0x103e4e350
```

**Atención:** El hecho de incorporar la dirección de memoria del objeto en el método `__repr__()` no es en absoluto obligatorio, ni siquiera necesario. Todo depende de los requerimientos que tengamos en el proyecto.

### Gestores de contexto

Otra de las aplicaciones interesantes de los métodos mágicos/especiales es la de los **gestores de contexto**. Un gestor de contexto permite aplicar una serie de *acciones a la entrada y a la salida* del bloque de código que engloba.

Hay dos métodos que son utilizados para implementar los gestores de contexto:

**`__enter__()`** Acciones que se llevan a cabo al entrar al contexto.

**`__exit__()`** Acciones que se llevan a cabo al salir del contexto.

Veamos un ejemplo en el que implementamos un gestor de contexto que **mide tiempos de ejecución**:

```
>>> from time import time

>>> class Timer():
...     def __enter__(self):
...         self.start = time()
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     def __exit__(self, exc_type, exc_value, exc_traceback):
...         # Omit exception handling
...         self.end = time()
...         exec_time = self.end - self.start
...         print(f'Execution time (seconds): {exec_time:.5f}')
...
```

Aunque en este caso no estamos haciendo uso de los parámetros en la función `__exit__()`, hacen referencia a una posible *excepción* (error) que se produzca en la ejecución del bloque de código que engloba el contexto. Los tres parámetros son:

1. `exc_type` indicando el tipo de la excepción.
2. `exc_value` indicando el valor (mensaje) de la excepción.
3. `exc_traceback` indicando la «traza» (pila) de llamadas que llevaron hasta la excepción.

Ahora podemos probar nuestro gestor de contexto con un ejemplo concreto. La forma de «activar» el contexto es usar la sentencia `with` seguida del símbolo que lo gestiona:

```
>>> with Timer():
...     for _ in range(1_000_000):
...         x = 2 ** 20
...
Execution time (seconds): 0.05283

>>> with Timer():
...     x = 0
...     for _ in range(1_000_000):
...         x += 2 ** 20
...
Execution time (seconds): 0.08749
```

Volviendo a nuestro ejemplo de los droides de StarWars, vamos a crear un gestor de contexto que «congele» un droide para resetear su distancia recorrida:

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.name = name
...         self.covered_distance = 0
...
...     def move_up(self, steps: int) -> None:
...         self.covered_distance += steps
...         print(f'Moving {steps} steps')
...

>>> class FrozenDroid:  # Gestor de contexto!
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     def __enter__(self, name: str):
...         self.droid = Droid(name)
...         return self.droid
...
...     def __exit__(self, *err):
...         self.droid.covered_distance = 0
...
```

Veamos este gestor de contexto en acción:

```
>>> with FrozenDroid() as droid:
...     droid.move_up(10)
...     droid.move_up(20)
...     droid.move_up(30)
...     print(droid.covered_distance)
...
Moving 10 steps
Moving 20 steps
Moving 30 steps
60

>>> droid.covered_distance  # Distancia reseteada!
0
```

### 6.2.5 Herencia

#### Nivel intermedio

La **herencia** consiste en **construir una nueva clase partiendo de una clase existente**, pero que añade o modifica ciertos aspectos. La herencia se considera una buena práctica de programación tanto para *reutilizar código* como para *realizar generalizaciones*.

---

**Nota:** Cuando se utiliza herencia, la clase derivada, de forma automática, puede usar todo el código de la clase base sin necesidad de copiar nada explícitamente.

---

[6] Iconos por Freepik.

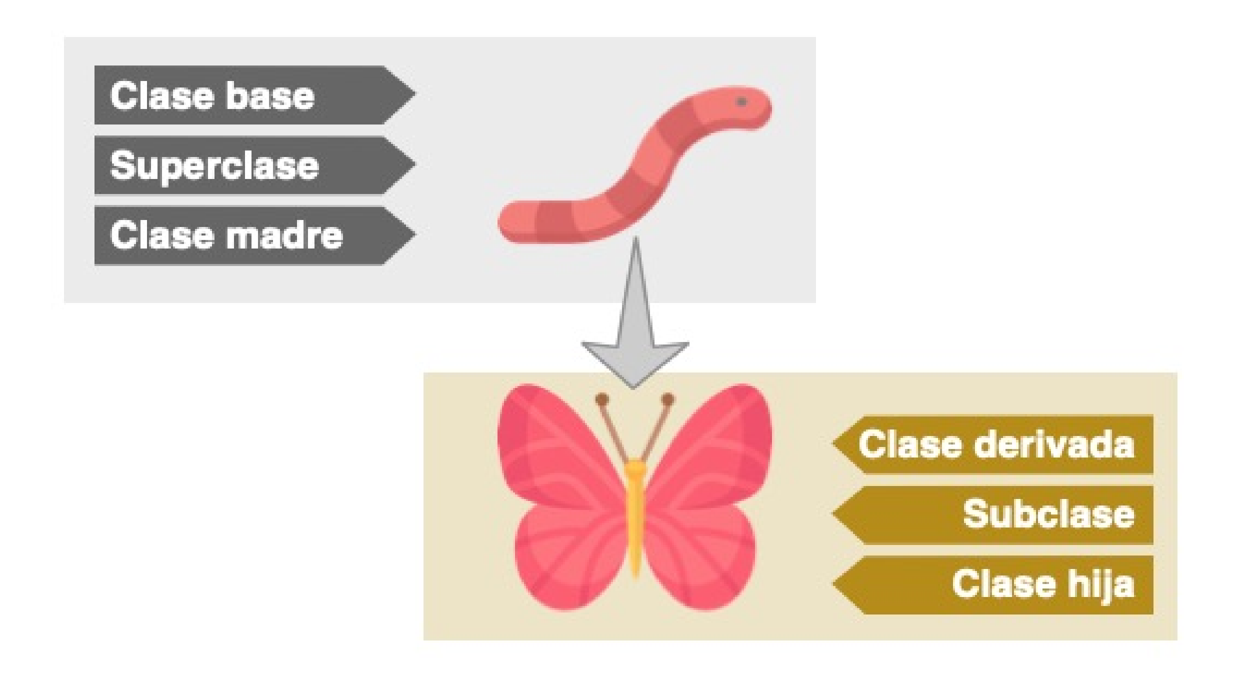



Figura 16: Nomenclatura de clases en la herencia[6]

### Heredar desde una clase base

Para que una clase «herede» de otra, basta con indicar la clase base entre paréntesis en la definición de la clase derivada.

Sigamos con el ejemplo galáctico: Una de las grandes categorías de droides en StarWars es la de droides de protocolo. Vamos a crear una herencia sobre esta idea:

```
>>> class Droid:
...     ''' Clase Base '''
...     pass
...

>>> class ProtocolDroid(Droid):
...     ''' Clase Derivada '''
...     pass
...

>>> issubclass(ProtocolDroid, Droid)  # comprobación de herencia
True

>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()
```

Vamos a añadir un par de métodos a la clase base, y analizar su comportamiento:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...

>>> class ProtocolDroid(Droid):
...     pass
...

>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()

>>> r2d2.switch_on()
Hi! I'm a droid. Can I help you?

>>> c3po.switch_on()  # método heredado de Droid
Hi! I'm a droid. Can I help you?

>>> r2d2.switch_off()
Bye! I'm going to sleep
```

### Sobreescribir un método

Como hemos visto, una clase derivada hereda todo lo que tiene su clase base. Pero en muchas ocasiones nos interesa modificar el comportamiento de esta herencia.

En el ejemplo anterior vamos a modificar el comportamiento del método `switch_on()` para la clase derivada:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...

>>> class ProtocolDroid(Droid):
...     def switch_on(self):
...         print("Hi! I'm a PROTOCOL droid. Can I help you?")
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()

>>> r2d2.switch_on()
Hi! I'm a droid. Can I help you?

>>> c3po.switch_on()  # método heredado pero sobreescrito
Hi! I'm a PROTOCOL droid. Can I help you?
```

### Añadir un método

La clase derivada puede, como cualquier otra clase «normal», añadir métodos que no estaban presentes en su clase base. En el siguiente ejemplo vamos a añadir un método `translate()` que permita a los *droides de protocolo* traducir cualquier mensaje:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...

>>> class ProtocolDroid(Droid):
...     def switch_on(self):
...         print("Hi! I'm a PROTOCOL droid. Can I help you?")
...
...     def translate(self, msg: str, *, from_lang: str) -> str:
...         ''' Translate from language to Human understanding '''
...         return f'{msg} means "ZASCA" in {from_lang}'

>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()

>>> c3po.translate('kiitos', from_lang='Huttese')  # idioma de Watoo
kiitos means "ZASCA" in Huttese

>>> r2d2.translate('kiitos', from_lang='Huttese')  # droide genérico no puede␣
↪traducir
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute 'translate'
```

Con esto ya hemos aportado una personalidad diferente a los droides de protocolo, a pesar

de que heredan de la clase genérica de droides de StarWars.

### Accediendo a la clase base

Cuando tenemos métodos (o atributos) definidos **con el mismo nombre** en la clase base y en la clase derivada (**colisión**) debe existir un mecanismo para diferenciarlos.

Para estas ocasiones Python nos ofrece `super()` como función para **acceder a métodos (o atributos) de la clase base**.

Este escenario es especialmente recurrente en el constructor de aquellas clases que heredan de otras y necesitan inicializar la clase base.

Veamos un ejemplo más elaborado con nuestros droides:

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.name = name
...

>>> class ProtocolDroid(Droid):
...     def __init__(self, name: str, languages: list[str]):
...         super().__init__(name)  # llamada al constructor de la clase base
...         self.languages = languages
...

>>> droid = ProtocolDroid('C-3PO', ['Ewokese', 'Huttese', 'Jawaese'])

>>> droid.name  # fijado en el constructor de la clase base
'C-3PO'

>>> droid.languages  # fijado en el constructor de la clase derivada
['Ewokese', 'Huttese', 'Jawaese']
```

### Herencia múltiple

#### Nivel avanzado

Aunque no está disponible en todos los lenguajes de programación, Python sí permite heredar de **múltiples clases base**.

Supongamos que queremos modelar la siguiente estructura de clases con *herencia múltiple*:

[5] Imágenes de los droides por StarWars Fandom.

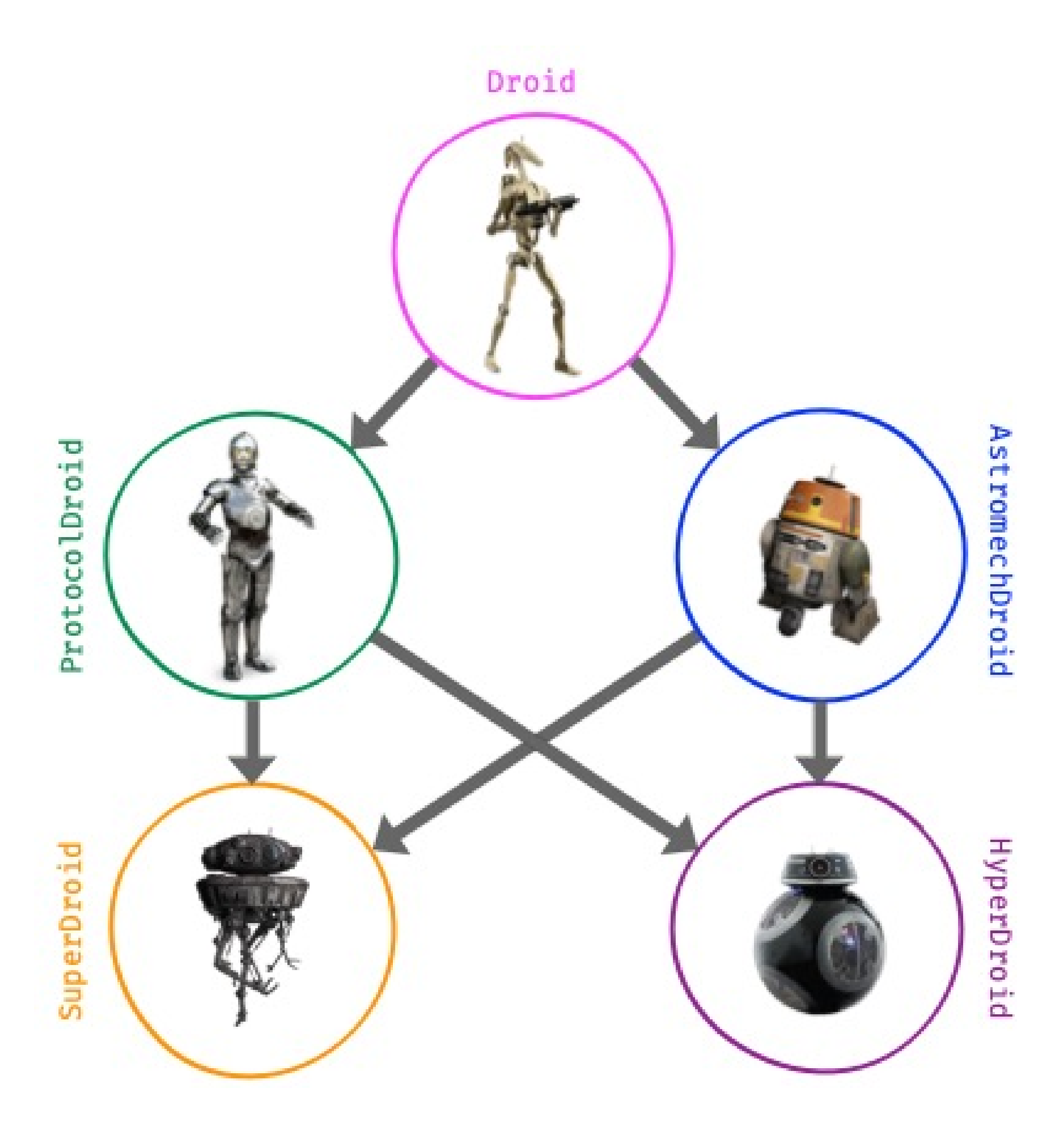



Figura 17: Ejemplo de herencia múltiple[5]

```
>>> class Droid:
...     def greet(self):
...         return 'Here a droid'
...

>>> class ProtocolDroid(Droid):
...     def greet(self):
...         return 'Here a protocol droid'
...

>>> class AstromechDroid(Droid):
...     def greet(self):
...         return 'Here an astromech droid'
...

>>> class SuperDroid(ProtocolDroid, AstromechDroid):
...     pass
...

>>> class HyperDroid(AstromechDroid, ProtocolDroid):
...     pass
```

**Prudencia:** El orden en el que especificamos varias clases base es importante.

Podemos comprobar esta herencia múltiple de la siguiente manera:

```
>>> issubclass(SuperDroid, (ProtocolDroid, AstromechDroid, Droid))
True

>>> issubclass(HyperDroid, (AstromechDroid, ProtocolDroid, Droid))
True
```

Veamos el resultado de la llamada a los métodos definidos para la jerarquía de droides:

```
>>> super_droid = SuperDroid()
>>> hyper_droid = HyperDroid()

>>> super_droid.greet()
'Here a protocol droid'

>>> hyper_droid.greet()
'Here an astromech droid'
```

Si en una clase se hace referencia a un método o atributo que no existe, Python lo buscará en todas sus clases base. Pero es posible que exista una *colisión* en caso de que el método

o el atributo buscado esté, a la vez, en varias clases base. En este caso, Python resuelve el conflicto a través del **orden de resolución de métodos**[4].

Todas las clases en Python disponen de un método especial llamado `mro()` «method resolution order» que devuelve una lista de las clases que se visitarían en caso de acceder a un método o a un atributo:

```
>>> SuperDroid.mro()
[__main__.SuperDroid,
 __main__.ProtocolDroid,
 __main__.AstromechDroid,
 __main__.Droid,
 object]
```

**Ver también:**

También se puede acceder a la misma información usando el atributo `__mro__`

Todos los objetos en Python heredan, en primera instancia, de `object`. Esto se puede comprobar con el correspondiente `mro()` de cada objeto:

```
>>> int.mro()
[int, object]

>>> str.mro()
[str, object]

>>> float.mro()
[float, object]

>>> tuple.mro()
[tuple, object]

>>> list.mro()
[list, object]

>>> bool.mro()  # Un booleano también es un entero!
[bool, int, object]
```

Lo anteriormente dicho puede explicarse igualmente a través del siguiente código:

```
>>> PY_TYPES = (int, str, float, tuple, list, bool)
>>> all(issubclass(_type, object) for _type in PY_TYPES)
True
```

[4] Viene del inglés «method resolution order» o `mro`.

## Mixins

Hay situaciones en las que nos interesa incorporar una clase base «independiente» de la jerarquía establecida, y sólo a efectos de **tareas auxiliares o transversales**. Esta aproximación podría ayudar a evitar *colisiones* en métodos o atributos reduciendo la ambigüedad que añade la herencia múltiple. A estas clases auxiliares se las conoce como **«mixins»**.

Veamos un ejemplo de un «mixin» para mostrar las variables de un objeto:

```
>>> class Instrospection:
...     def dig(self):
...         print(vars(self))  # vars devuelve las variables del argumento
...
... class Droid(Instrospection):
...     pass
...

>>> droid = Droid()

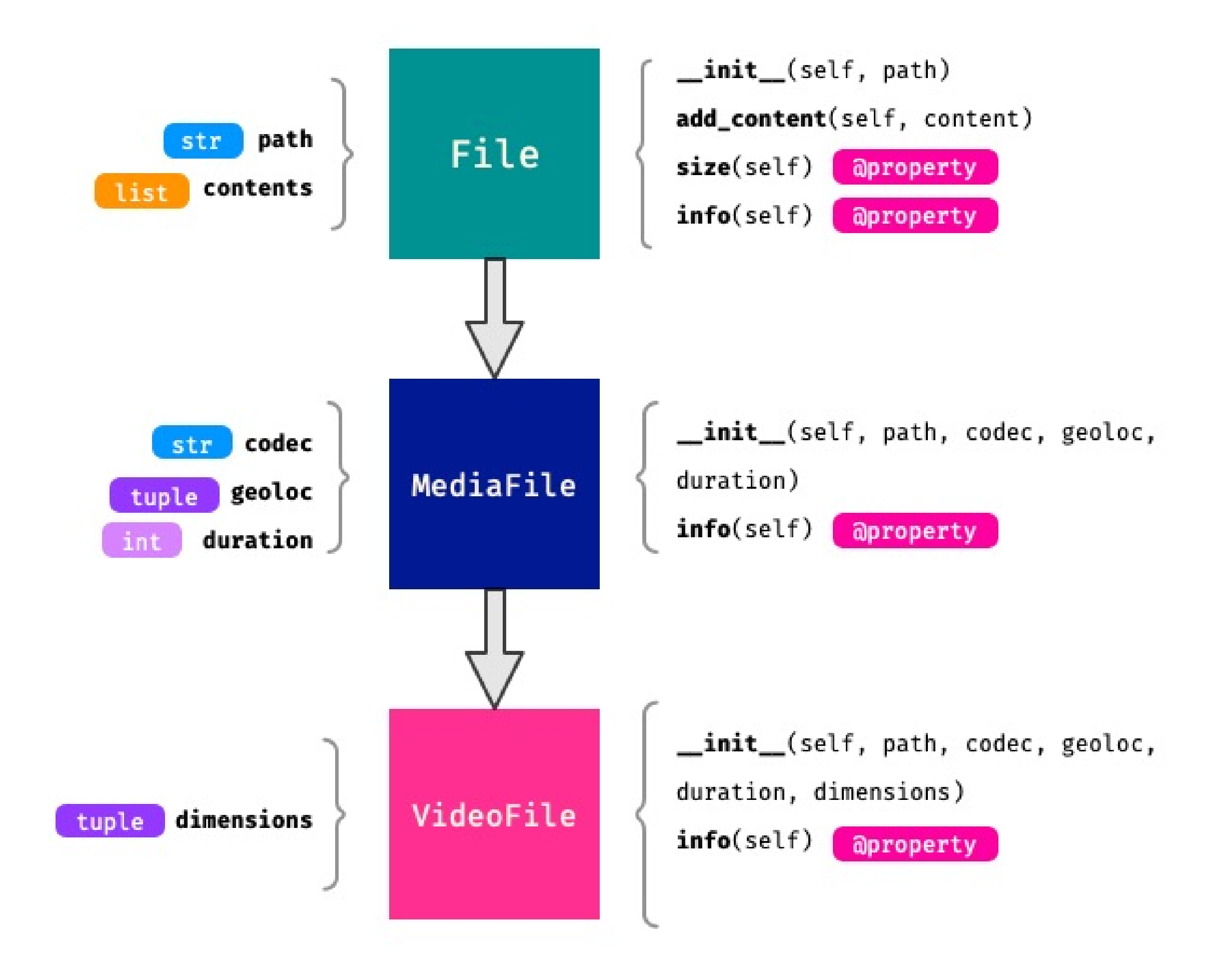
>>> droid.code = 'DN-LD'
>>> droid.num_feet = 2
>>> droid.type = 'Power Droid'

>>> droid.dig()
{'code': 'DN-LD', 'num_feet': 2, 'type': 'Power Droid'}
```

---

**Ejercicio**

Cree el siguiente escenario de herencia de clases en Python que representa distintos tipos de ficheros en un sistema:




Notas:

- El atributo `size` debe devolver el número total de caracteres sumando las longitudes de los elementos del atributo `contents`.
- El atributo `info` de cada clase debe hacer uso del atributo `info` de su clase base para conformar las salida final.

---

## Agregación y composición

Aunque la herencia de clases nos permite modelar una gran cantidad de casos de uso en términos de «**is-a**» (*es un*), existen muchas otras situaciones en las que la agregación o la composición son una mejor opción. En este caso una clase se compone de otras clases: hablamos de una relación «**has-a**» (*tiene un*).

Hay una sutil diferencia entre agregación y composición:

- La **composición** implica que el objeto utilizado no puede «funcionar» sin la presencia de su propietario.

- La **agregación** implica que el objeto utilizado puede funcionar por sí mismo.




Figura 18: Agregación vs. Composición[6]

Veamos un ejemplo de **agregación** en el que añadimos una herramienta a un droide:

```
>>> class Tool:
...     def __init__(self, name: str):
...         self.name = name
...
...     def __str__(self):
...         return self.name.upper()
...
... class Droid:
...     def __init__(self, name: str, serial_number: int, tool: Tool):
...         self.name = name
...         self.serial_number = serial_number
...         self.tool = tool  # agregación
...
...     def __str__(self):
...         return f'Droid {self.name} armed with a {self.tool}'
...

>>> lighter = Tool('lighter')
>>> bb8 = Droid('BB-8', 48050989085439, lighter)

>>> print(bb8)
Droid BB-8 armed with a LIGHTER
```

### 6.2.6 Estructuras mágicas

Obviamente no existen estructuras mágicas, pero sí que hay estructuras de datos que deben implementar ciertos métodos mágicos (o especiales) para desarrollar su comportamiento.

En este apartado veremos algunos de ellos.

#### Secuencias

Una **secuencia** en Python es un objeto en el que podemos acceder a cada uno de sus elementos a través de un índice, así como calcular su longitud total.

```
obj[0] ⟷ obj.__getitem__(0)
obj[1] = value ⟷ obj.__setitem__(1, value)
len(obj) ⟷ obj.__len__()
```

Figura 19: Métodos mágicos asociados con las secuencias

Como ejemplo, podemos asumir que los droides de StarWars están ensamblados con distintas partes/componentes. Veamos una implementación de este escenario:

```
>>> class Droid:
...     def __init__(self, name: str, parts: list[str]):
...         self.name = name
...         self.parts = parts
...
...     def __setitem__(self, index: int, part: str) -> None:
...         self.parts[index] = part
...
...     def __getitem__(self, index: int) -> str:
...         return self.parts[index]
...
...     def __len__(self):
...         return len(self.parts)
...
```

Ahora podemos poner instanciar la clase anterior y probar su comportamiento:

```
>>> droid = Droid('R2-D2', ['Radar Eye', 'Pocket Vent', 'Battery Box'])

>>> droid.parts
```

(continué en la próxima página)

(proviene de la página anterior)

```
['Radar Eye', 'Pocket Vent', 'Battery Box']

>>> droid[0]  # __getitem__(0)
'Radar Eye'
>>> droid[1]  # __getitem__(1)
'Pocket Vent'
>>> droid[2]  # __getitem__(2)
'Battery Box'

>>> droid[1] = 'Holographic Projector'  # __setitem__()

>>> droid.parts
['Radar Eye', 'Holographic Projector', 'Battery Box']

>>> len(droid)  # __len__()
3
```

**Ejercicio**

Cree una clase `InfiniteList` que permita utilizar una lista sin tener límites, es decir, evitando un `IndexError`. Por ejemplo, si la lista tiene 10 elementos, y asignamos un valor al elemento en el índice 20, esto no daría un error, sino que haría ampliar la lista hasta el valor 20, rellenando los valores en blanco con un valor de relleno que por defecto es `None`.

### Diccionarios

Los métodos `__getitem__()` y `__setitem()__` también se pueden aplicar para obtener o fijar valores en un estructura tipo **diccionario**. La diferencia es que en vez de manejar un índice manejamos una clave.

Retomando el ejemplo anterior de las partes de un droide vamos a plantear que **cada componente tiene asociado una versión**, lo que nos proporciona una estructura de tipo diccionario con clave (nombre de la parte) y valor (versión de la parte):

```
>>> class Droid:
...     def __init__(self, name: str, parts: dict[str, float]):
...         self.name = name
...         self.parts = parts
...
...     def __setitem__(self, part: str, version: float) -> None:
...         self.parts[part] = version
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     def __getitem__(self, part: str) -> float:
...         return self.parts.get(part)
...
...     def __len__(self):
...         return len(self.parts)
```

Ahora podremos instanciar la clase anterior y comprobar su comportamiento:

```
>>> droid = Droid(
...     'R2-D2',
...     {
...         'Radar Eye': 1.1,
...         'Pocket Vent': 3.0,
...         'Battery Box': 2.8
...     }
... )

>>> droid.parts
{'Radar Eye': 1.1, 'Pocket Vent': 3.0, 'Battery Box': 2.8}

>>> droid['Radar Eye']
1.1
>>> droid['Pocket Vent']
3.0
>>> droid['Battery Box']
2.8

>>> droid['Pocket Vent'] = 3.1

>>> droid.parts
{'Radar Eye': 1.1, 'Pocket Vent': 3.1, 'Battery Box': 2.8}

>>> len(droid)
3
```

## Iterables

### Nivel avanzado

Un objeto en Python se dice **iterable** si implementa el **protocolo de iteración**. Este protocolo permite «entregar» un valor de cada vez en forma de secuencia.

Hay muchos tipos de datos iterables en Python que ya hemos visto: *cadenas de texto, listas, tuplas, conjuntos, diccionarios, etc.*

Para ser un **objeto iterable** sólo es necesario implementar el método mágico `__iter__()`.

Este método debe proporcionar una referencia al **objeto iterador** que es quien se encargará de desarrollar el protocolo de iteración a través del método mágico `__next__()`.

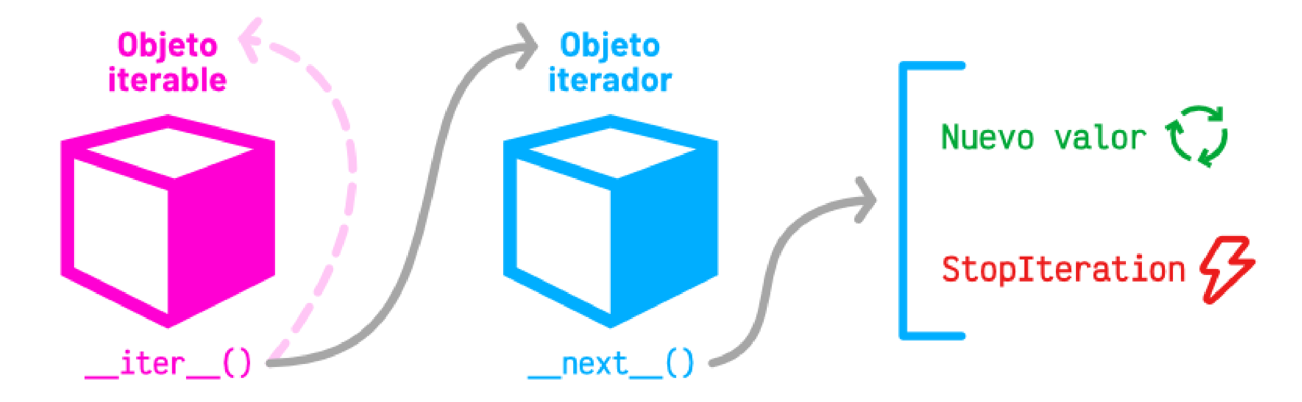



Figura 20: Protocolo de iteración

**Truco:** **Spoiler**: Un objeto iterable también puede ser su propio iterador.

Veamos un ejemplo del universo StarWars. Vamos a partir de un modelo muy sencillo de droide:

```
>>> class Droid:
...     def __init__(self, serial: str):
...         self.serial = serial * 5  # just for fun!
...
...     def __str__(self):
...         return f'Droid: SN={self.serial}'
```

Vamos a implementar una factoría de droides (Geonosis) como un iterable:

```
>>> class Geonosis:
...     def __init__(self, num_droids: int):
...         self.num_droids = num_droids
...         self.pointer = 0
...
...     def __iter__(self):
...         # El iterador es el propio objeto!
...         return self
...
...     def __next__(self):
...         # Protocolo de iteración
...         if self.pointer >= self.num_droids:
...             raise StopIteration
...         droid = Droid(str(self.pointer))
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         self.pointer += 1
...         return droid
...
```

Ahora podemos recorrer el iterable y obtener los droides que genera la factoría:

```
>>> for droid in Geonosis(10):
...     print(droid)
...
Droid: SN=00000
Droid: SN=11111
Droid: SN=22222
Droid: SN=33333
Droid: SN=44444
Droid: SN=55555
Droid: SN=66666
Droid: SN=77777
Droid: SN=88888
Droid: SN=99999
```

Cuando utilizamos un bucle `for` para recorrer los elementos de un iterable, ocurren varias cosas:

1. Se obtiene el objeto iterador del iterable.
2. Se hacen llamadas sucesivas a `next()` sobre dicho iterador para obtener cada elemento del iterable.
3. Se para la iteración cuando el iterador lanza la excepción `StopIteration`.

## Iterables desde fuera

Ahora que conocemos las interiodades de los iterables, podemos ver qué ocurre si los usamos desde un enfoque más funcional.

En primer lugar hay que controlar el uso de los **métodos mágicos en el protocolo de iteración**:

- `__iter()__` se invoca cuando se hace uso de la función `iter()`.
- `__next()__` se invoca cuando se hace uso de la función `next()`.

Si esto es así, podríamos generar droides de una forma algo «peculiar»:

```
>>> factory = Geonosis(3)

>>> factory_iterator = iter(factory)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> next(factory_iterator)
Droid: SN=00000
>>> next(factory_iterator)
Droid: SN=11111
>>> next(factory_iterator)
Droid: SN=22222

>>> next(factory_iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Se da la circunstancia de que, en este caso, **no tenemos que crear el iterador** para poder obtener nuevos elementos:

```
>>> next(Geonosis(3))
Droid: SN=00000
```

Esto básicamente se debe a que **el iterador es el propio iterable**:

```
>>> geon_iterable = Geonosis(3)
>>> geon_iterator = iter(geon_iterable)

>>> geon_iterable is geon_iterator
True
```

Otra característica importante es que **los iterables se agotan**. Lo podemos comprobar con el siguiente código:

```
>>> geon = Geonosis(3)

>>> for droid in geon:
...     print(droid)
...
Droid: SN=00000
Droid: SN=11111
Droid: SN=22222

>>> for droid in geon:
...     print(droid)
... # Salida vacía!
```

**Ejercicio**

pycheck: **fibonacci_iterable**

### Usando un iterador externo

Hasta ahora hemos analizado el escenario en el que el objeto iterable coincide con el objeto iterador, pero esto no tiene por qué ser siempre así.

Supongamos ahora que queremos implementar un **mercado de droides de protocolo** que debe ser un iterable y devolver cada vez un droide de protocolo. Veamos esta aproximación **usando un iterador externo**:

```
>>> class Droid:
...     def __init__(self, name: str):
...         self.name = name
...
...     def __repr__(self):
...         return f'Droid: {self.name}'
...

>>> class ProtocolDroidMarket:
...     DROID_MODELS = ('C-3PO', 'K-3PO', 'R-3PO', 'RA-7',
...                     'TC-14', 'TC-4', '4-LOM')
...
...     def __init__(self):
...         self.droids = [Droid(name) for name in ProtocolDroidMarket.DROID_MODELS]
...
...     def __iter__(self):
...         return ProtocolDroidMarketIterator(self)
...

>>> class ProtocolDroidMarketIterator:
...     def __init__(self, market: ProtocolDroidMarket):
...         self.market = market
...         self.pointer = 0
...
...     def __next__(self):
...         if self.pointer >= len(self.market.droids):
...             raise StopIteration
...         droid = self.market.droids[self.pointer]
...         self.pointer += 1
...         return droid
...
```

Probamos ahora el código anterior recorriendo todos los droides que están disponibles en el mercado:

```
>>> market = ProtocolDroidMarket()

>>> for droid in market:
...     print(droid)
...
Droid: C-3PO
Droid: K-3PO
Droid: R-3PO
Droid: RA-7
Droid: TC-14
Droid: TC-4
Droid: 4-LOM
```

**Consejo:** Esta aproximación puede ser interesante cuando no queremos mezclar el código del iterador con la lógica del objeto principal.

### Ejemplos de iterables

Vamos a analizar herramientas ya vistas – entendiendo mejor su funcionamiento interno – en base a lo que ya sabemos sobre iterables.

**Enumeración**:

```
>>> tool = enumerate([1, 2, 3])

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # es su propio iterador!
True

>>> next(tool)
(0, 1)
>>> next(tool)
(1, 2)
>>> next(tool)
(2, 3)

>>> next(tool)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Rangos**:

```
>>> tool = range(1, 4)

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # usa otro iterador!
False

>>> tool_iterator = iter(tool)

>>> tool_iterator
<range_iterator at 0x1100e6d60>

>>> next(tool_iterator)
1
>>> next(tool_iterator)
2
>>> next(tool_iterator)
3

>>> next(tool_iterator)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Nota:** Los objetos de tipo `range` representan una secuencia inmutable de números. La ventaja de usar este tipo de objetos es que siempre se usa una cantidad fija (y pequeña) de memoria, independientemente del rango que represente (ya que solamente necesita almacenar los valores para `start`, `stop` y `step`, y calcula los valores intermedios a medida que los va necesitando).

**Invertido**:

```
>>> tool = reversed([1, 2, 3])

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # es su propio iterador!
True

>>> next(tool)
3
>>> next(tool)
```

(continué en la próxima página)

(proviene de la página anterior)

```
2
>>> next(tool)
1

>>> next(tool)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Comprimir**:

```
>>> tool = zip([1, 2], [3, 4])

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # es su propio iterador!
True

>>> next(tool)
(1, 3)
>>> next(tool)
(2, 4)

>>> next(tool)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Generadores**:

```
>>> def seq(n):
...     for i in range(1, n+1):
...         yield i
...

>>> tool = seq(3)

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # es su propio iterador!
True

>>> next(tool)
```

(continué en la próxima página)

(proviene de la página anterior)

```
1
>>> next(tool)
2
>>> next(tool)
3

>>> next(tool)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Ver también:**

Esto mismo se puede aplicar a expresiones generadoras.

**Listas**:

```
>>> tool = [1, 2, 3]

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # usa otro iterador!
False

>>> tool_iterator = iter(tool)

>>> tool_iterator
<list_iterator at 0x1102492d0>

>>> next(tool_iterator)
1
>>> next(tool_iterator)
2
>>> next(tool_iterator)
3

>>> next(tool_iterator)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Tuplas**:

```
>>> tool = tuple([1, 2, 3])
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # usa otro iterador!
False

>>> tool_iterator = iter(tool)

>>> tool_iterator
<tuple_iterator at 0x107255a50>

>>> next(tool_iterator)
1
>>> next(tool_iterator)
2
>>> next(tool_iterator)
3

>>> next(tool_iterator)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Cadenas de texto**:

```
>>> tool = 'abc'

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # usa otro iterador!
False

>>> tool_iterator = iter(tool)

>>> tool_iterator
<str_ascii_iterator at 0x1078da7d0>

>>> next(tool_iterator)
'a'
>>> next(tool_iterator)
'b'
>>> next(tool_iterator)
'c'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> next(tool_iterator)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Diccionarios**:

```
>>> tool = dict(a=1, b=1)

>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # usa otro iterador!
False

>>> tool_iterator = iter(tool)

>>> tool_iterator
<dict_keyiterator at 0x1070200e0>

>>> next(tool_iterator)
'a'
>>> next(tool_iterator)
'b'

>>> next(tool_iterator)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el caso de los diccionarios existen varios iteradores disponibles:

```
>>> iter(tool.keys())
<dict_keyiterator at 0x107849ad0>

>>> iter(tool.values())
<dict_valueiterator at 0x1102aab10>

>>> iter(tool.items())
<dict_itemiterator at 0x107df6ac0>
```

**Conjuntos**:

```
>>> tool = set([1, 2, 3])
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> iter(tool) is not None  # es iterable!
True

>>> iter(tool) == tool  # usa otro iterador!
False

>>> tool_iterator = iter(tool)

>>> tool_iterator
<set_iterator at 0x10700e900>

>>> next(tool_iterator)
1
>>> next(tool_iterator)
2
>>> next(tool_iterator)
3

>>> next(tool_iterator)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**Ficheros**:

```
>>> f = open('data.txt')

>>> iter(f) is not None  # es iterable!
True

>>> iter(f) == f  # es su propio iterador!
True

>>> next(f)
'1\n'
>>> next(f)
'2\n'
>>> next(f)
'3\n'

>>> next(f)  # protocolo de iteración!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Todos las herramientas anteriores las podemos resumir en la siguiente tabla:

| Herramienta | Es iterable | Es iterador | Múltiples iteradores |
|---|---|---|---|
| enumerate() | ✓ | ✓ | |
| reversed() | ✓ | ✓ | |
| zip() | ✓ | ✓ | |
| generator | ✓ | ✓ | |
| file | ✓ | ✓ | |
| list() | ✓ | | |
| tuple() | ✓ | | |
| str() | ✓ | | |
| dict() | ✓ | | ✓ |
| set() | ✓ | | |
| range() | ✓ | | |

### 6.2.7 Estructura de una clase

Durante toda la sección hemos analizado con detalle los distintos componentes que forman una clase en Python. Pero cuando todo esto lo ponemos junto puede suponer un pequeño caos organizativo.

Aunque no existe ninguna indicación formal de la estructura de una clase, podríamos establecer el siguiente formato como guía de estilo:

```
>>> class OrganizedClass:
...     '''Descripción de la clase'''
...
...     # Constructor
...
...     # Métodos de instancia
...
...     # Propiedades
...
...     # Métodos mágicos
...
...     # Decoradores
...
...     # Métodos de clase
...
...     # Métodos estáticos
...
...     ...
...
```

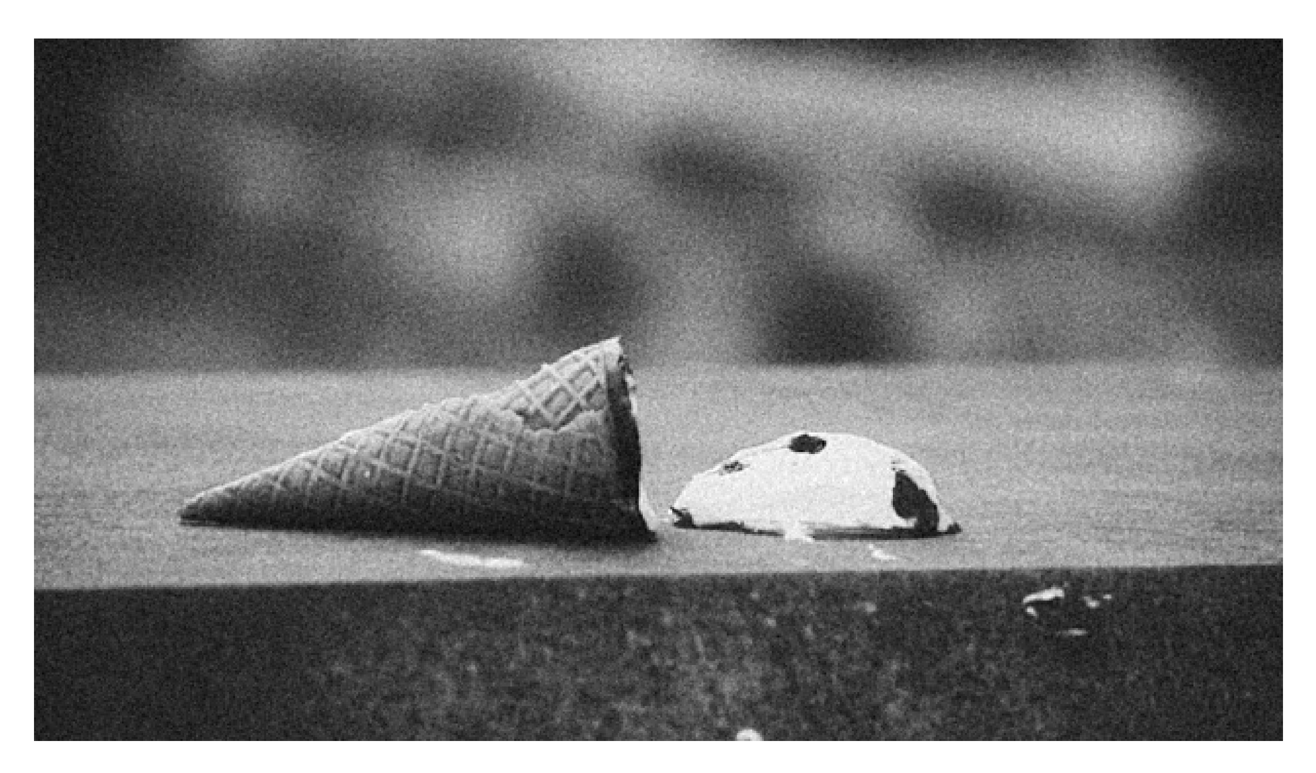
### EJERCICIOS DE REPASO

1. Escriba una clase `Date` que represente una fecha.
2. Escriba una clase `DNA` que represente una secuencia de ADN.
3. Escriba una clase `IntegerStack` que represente una pila de valores enteros.
3. Escriba una clase `IntegerQueue` que represente una cola de valores enteros.

### AMPLIAR CONOCIMIENTOS

- Supercharge Your Classes With Python super()
- Inheritance and Composition: A Python OOP Guide
- OOP Method Types in Python: @classmethod vs @staticmethod vs Instance Methods
- Intro to Object-Oriented Programming (OOP) in Python
- Pythonic OOP String Conversion: __repr__ vs __str__
- @staticmethod vs @classmethod in Python
- Modeling Polymorphism in Django With Python
- Operator and Function Overloading in Custom Python Classes
- Object-Oriented Programming (OOP) in Python 3
- Why Bother Using Property Decorators in Python?

## 6.3 Excepciones



Una **excepción** es el bloque de código que se lanza cuando se produce un **error** en la ejecución de un programa Python.[1]

De hecho ya hemos visto algunas de estas excepciones: accesos fuera de rango a listas o tuplas, accesos a claves inexistentes en diccionarios, etc. Cuando ejecutamos código que podría fallar bajo ciertas circunstancias, necesitamos también manejar, de manera adecuada, las excepciones que se generan.

### 6.3.1 Manejando errores

Si una excepción ocurre en algún lugar de nuestro programa y no es capturada en ese punto, va subiendo (burbujeando) hasta que es capturada en alguna función que ha hecho la llamada. Si en toda la «pila» de llamadas no existe un control de la excepción, Python muestra un mensaje de error con información adicional:

```
>>> def intdiv(a: int, b: int) -> int:
...     return a // b
...

>>> intdiv(3, 0)
Traceback (most recent call last):
```

(continué en la próxima página)

[1] Foto original por Sarah Kilian en Unsplash.

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in intdiv
ZeroDivisionError: integer division or modulo by zero
```

Para manejar (capturar) las excepciones podemos usar un bloque de código con las palabras reservadas `try` and `except`:

```
>>> def intdiv(a: int, b: int) -> int:
...     try:
...         return a // b
...     except:
...         print('Please do not divide by zero...')
...

>>> intdiv(3, 0)
Please do not divide by zero...
```

Aquel código que se encuentre dentro del bloque `try` se ejecutará normalmente siempre y cuando no haya un error. Si se produce una excepción, ésta será capturada por el bloque `except`, ejecutándose el código que contiene.

---

**Consejo:** No es una buena práctica usar un bloque `except` sin indicar el **tipo de excepción** que estamos gestionando, no sólo porque puedan existir varias excepciones que capturar sino porque, como dice el *Zen de Python*: «explícito» es mejor que «implícito».

---

La traza o **pila de llamadas** («Traceback» en inglés) se muestra cada vez que se produce una excepción en nuestro programa y contiene todas las llamadas que han intervenido en el proceso. Dependiendo de lo anidado que esté el error, tendremos una traza más o menos grande:

```
>>> def intdiv(a: int, b: int) -> int:
...     return a // b
...
>>> def arithmetics():
...     return intdiv(3, 0)
...
>>> def manage():
...     return arithmetics()
...
>>> def main():
...     return manage()
...

>>> main()
```

(continué en la próxima página)

(proviene de la página anterior)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in main
  File "<stdin>", line 2, in manage
  File "<stdin>", line 2, in arithmetics
  File "<stdin>", line 2, in intdiv
ZeroDivisionError: integer division or modulo by zero
```

## Especificando excepciones

En el siguiente ejemplo mejoraremos el código anterior, capturando distintos tipos de excepciones predefinidas:

- `TypeError` por si los operandos no permiten la división.
- `ZeroDivisionError` por si el denominador es cero.
- `Exception` para cualquier otro error que se pueda producir.

Veamos su implementación:

```
>>> def intdiv(a, b):
...     try:
...         result = a // b
...     except TypeError:
...         print('Check operands. Some of them seems strange...')
...     except ZeroDivisionError:
...         print('Please do not divide by zero...')
...     except Exception:
...         print('Ups. Something went wrong...')
...

>>> intdiv(3, 0)
Please do not divide by zero...

>>> intdiv(3, '0')
Check operands. Some of them seems strange...
```

### Excepciones predefinidas

Las excepciones predefinidas en Python cubren un amplio rango de posibilidades y *no hace falta importarlas previamente*. Se pueden usar directamente.

Conocerlas es importante ya que nos permitirá gestionar mejor los posibles errores y dar respuesta a situaciones inesperadas. Veamos a continuación algunas de las más relevantes:

| Excepción | Significado | Ejemplo |
|---|---|---|
| AttributeError | Referencia de atributo inexistente | 'hello'.splik() |
| IndexError | Subíndice de secuencia fuera de rango | (2, 3)[5] |
| KeyError | Clave de diccionario no encontrada | {0:1, 1:2}[2] |
| NotImplementedError | La operación debe ser implementada | |
| OSError | Error al usar una función del sistema operativo (E/S) | open('null.txt') |
| RecursionError | Alcanzado el máximo nivel de recursión | |
| StopIteration | Fin del protocolo de iteración | |
| TypeError | Operación sobre un objeto de tipo inapropiado | 'x' / 3 |
| ValueError | Operación sobre un objeto de tipo correcto pero valor inapropiado | int('x') |
| ZeroDivisionError | Segundo argumento de división o módulo es cero | 1 / 0 |

### Agrupando excepciones

Si nos interesa tratar distintas excepciones con el mismo comportamiento, es posible agruparlas en una única línea:

```
>>> def intdiv(a, b):
...     try:
...         result = a // b
...     except (TypeError, ZeroDivisionError):
...         print('Check operands: Some of them caused errors...')
...     except Exception:
...         print('Ups. Something went wrong...')
...

>>> intdiv(3, 0)
Check operands: Some of them caused errors...
```

### Variantes en el tratamiento

Python proporciona la cláusula `else` para saber que todo ha ido bien y que no se ha lanzado ninguna excepción. Esto es relevante a la hora de manejar los errores.

De igual modo, tenemos a nuestra disposición la cláusula `finally` que se ejecuta siempre, independientemente de si ha habido o no ha habido error.

Veamos un ejemplo de ambos:

```
>>> values = [4, 2, 7]

>>> try:
...     r = values[3]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Error: Index not in list
Have a good day!

>>> try:
...     r = values[2]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Your wishes are my command: 7
Have a good day!
```

---

**Ejercicio**

Sabiendo que `ValueError` es la excepción que se lanza cuando no podemos convertir una cadena de texto en su valor numérico, escriba una función `getint()` que lea un valor entero del usuario y lo devuelva, iterando mientras el valor no sea correcto.

Ejecución a modo de ejemplo:

```
Give me an integer number: ten
Not a valid integer. Try it again!
Give me an integer number: diez
```

(continué en la próxima página)

(proviene de la página anterior)

```
Not a valid integer. Try it again!
Give me an integer number: 10
```

*Trate de implementar tanto la versión recursiva como la versión iterativa.*

## Mostrando los errores

Además de capturar las excepciones podemos mostrar sus mensajes de error asociados. Para ello tendremos que hacer uso de la palabra reservada `as` junto a un nombre de variable que contendrá el objeto de la excepción.

Veamos este comportamiento siguiendo con el ejemplo anterior:

```
>>> try:
...     print(values[3])
... except IndexError as err:
...     print(err)
...
list index out of range
```

Una vez con la excepción capturada, ya podemos «elaborar» un poco más el mensaje de salida:

```
>>> try:
...     print(values[3])
... except IndexError as err:
...     print(f'Something went wrong: {err}')
...
Something went wrong: list index out of range
```

**Ver también:**

Este «alias» también es posible aplicarlo cuando *agrupamos excepciones*.

## Elevando excepciones

Es habitual que nuestro programa tenga que lanzar (elevar o levantar) una excepción (predefinida o propia). Para ello tendremos que hacer uso de la sentencia `raise`.

Supongamos una función que suma dos valores enteros. En el caso de que alguno de los operandos no sea entero, elevaremos una excepción indicando esta circunstancia:

```
>>> def _sum(a: int, b: int) -> int:
...     if isinstance(a, int) and isinstance(b, int):
...         return a + b
...     raise TypeError('Operands must be integers')
...

>>> _sum(4, 3)  # todo normal
7

>>> _sum('x', 'y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in _sum
TypeError: Operands must be integers
```

## Jerarquía de excepciones

Todas las excepciones predefinidas en Python heredan de la clase `Exception` y de la clase `BaseException` (más allá de heredar, obviamente, de `object`).

Podemos visitar algunas *excepciones predefinidas* y comprobar este comportamiento:

```
>>> TypeError.mro()
[TypeError, Exception, BaseException, object]

>>> ZeroDivisionError.mro()
[ZeroDivisionError, ArithmeticError, Exception, BaseException, object]

>>> IndexError.mro()
[IndexError, LookupError, Exception, BaseException, object]

>>> FileNotFoundError.mro()
[FileNotFoundError, OSError, Exception, BaseException, object]
```

A continuación se detalla la **jerarquía completa de excepciones predefinidas** en Python:

```
BaseException
□□□ BaseExceptionGroup
□□□ GeneratorExit
□□□ KeyboardInterrupt
□□□ SystemExit
□□□ Exception
    □□□ ArithmeticError
    □    □□□ FloatingPointError
    □    □□□ OverflowError
```

(continué en la próxima página)

(proviene de la página anterior)

```
□    □□□ ZeroDivisionError
□□□ AssertionError
□□□ AttributeError
□□□ BufferError
□□□ EOFError
□□□ ExceptionGroup [BaseExceptionGroup]
□□□ ImportError
□    □□□ ModuleNotFoundError
□□□ LookupError
□    □□□ IndexError
□    □□□ KeyError
□□□ MemoryError
□□□ NameError
□    □□□ UnboundLocalError
□□□ OSError
□    □□□ BlockingIOError
□    □□□ ChildProcessError
□    □□□ ConnectionError
□    □    □□□ BrokenPipeError
□    □    □□□ ConnectionAbortedError
□    □    □□□ ConnectionRefusedError
□    □    □□□ ConnectionResetError
□    □□□ FileExistsError
□    □□□ FileNotFoundError
□    □□□ InterruptedError
□    □□□ IsADirectoryError
□    □□□ NotADirectoryError
□    □□□ PermissionError
□    □□□ ProcessLookupError
□    □□□ TimeoutError
□□□ ReferenceError
□□□ RuntimeError
□    □□□ NotImplementedError
□    □□□ RecursionError
□□□ StopAsyncIteration
□□□ StopIteration
□□□ SyntaxError
□    □□□ IndentationError
□         □□□ TabError
□□□ SystemError
□□□ TypeError
□□□ ValueError
□    □□□ UnicodeError
□         □□□ UnicodeDecodeError
□         □□□ UnicodeEncodeError
```

(continué en la próxima página)

(proviene de la página anterior)

```
    □         □□□ UnicodeTranslateError
    □□□ Warning
        □□□ BytesWarning
        □□□ DeprecationWarning
        □□□ EncodingWarning
        □□□ FutureWarning
        □□□ ImportWarning
        □□□ PendingDeprecationWarning
        □□□ ResourceWarning
        □□□ RuntimeWarning
        □□□ SyntaxWarning
        □□□ UnicodeWarning
        □□□ UserWarning
```

**Truco:** Si capturamos una clase base estaremos capturando todas sus clases derivadas. Esto no es cierto a la inversa.

## 6.3.2 Excepciones propias

### Nivel avanzado

Python ofrece una gran cantidad de excepciones predefinidas. Hasta ahora hemos visto cómo gestionar y manejar este tipo de excepciones. Pero hay ocasiones en las que nos puede interesar crear nuestras propias excepciones. Para ello simplemente tendremos que crear una clase *heredando* de `Exception`, la clase base para todas las excepciones.

Veamos un ejemplo en el que creamos una excepción propia controlando que el valor sea un número entero:

```
>>> class NotIntError(Exception):
...     pass
...

>>> values = (4, 7, 2.11, 9)

>>> for value in values:
...     if not isinstance(value, int):
...         raise NotIntError(value)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
NotIntError: 2.11
```

Hemos usado la sentencia `raise` para *elevar esta excepción*, que podría ser controlada en un nivel superior mediante un bloque `try` - `except`.

---

**Nota:** Para crear una excepción propia basta con crear una clase vacía. No es necesario incluir código más allá de un `pass`.

---

### Mensaje personalizado

Podemos personalizar la excepción propia añadiendo un mensaje como **valor por defecto**. Siguiendo el ejemplo anterior, veamos cómo introducimos esta información:

```
>>> class NotIntError(Exception):
...     def __init__(self, message='This module only works with integers. Sorry!'):
...         super().__init__(message)
...

>>> raise NotIntError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotIntError: This module only works with integers. Sorry!
```

Supongamos que queremos ir un paso más allá e **incorporar en el mensaje de la excepción el propio valor** que está generando el error:

```
>>> class NotIntError(Exception):
...     def __init__(self, value, *, message='This module only works with integers.␣
↪Sorry!'):
...         self.value = value
...         self.message = message
...         super().__init__(self.message)
...
...     def __str__(self):
...         return f'{self.value} -> {self.message}'
...

>>> raise NotIntError(2.11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotIntError: 2.11 -> This module only works with integers. Sorry!
```

Y con esta misma configuración podemos **modificar el mensaje por defecto**:

```
>>> raise NotIntError(2.11, message='Please use integers!')
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
NotIntError: 2.11 -> Please use integers!
```

**Nota:** Una excepción propia no es más que una clase ordinaria y, por tanto, admite cualquier tipo de parámetro en su constructor y resto de métodos. Si se usa con sentido puede ser una poderosa herramienta.

**No siempre es necesario** implementar el método `__str__()`. Veamos una reescritura del código anterior:

```
>>> class NotIntError(Exception):
...     def __init__(self, value, *, message='This module only works with integers.␣
↪Sorry!'):
...         err_info = f'{value} -> {message}'
...         super().__init__(err_info)
...
```

Nos estamos apoyando en el hecho de que `NotIntError` hereda de `Exception` y esta clase base ya dispone de un método `__str__()`. Podemos comprobar que su comportamiento es igual que antes:

```
>>> raise NotIntError(2.11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotIntError: 2.11 -> This module only works with integers. Sorry!

>>> raise NotIntError(2.11, message='Please use integers!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotIntError: 2.11 -> Please use integers!
```

### 6.3.3 Aserciones

Si hablamos de control de errores hay que citar una sentencia en Python denominada `assert`. Esta sentencia nos permite comprobar si se están cumpliendo las «expectativas» de nuestro programa, y en caso contrario, lanza una excepción informativa.

Su sintaxis es muy simple. Únicamente tendremos que indicar una expresión de comparación después de la sentencia:

```
>>> result = 10
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> assert result > 0

>>> print(result)
10
```

En el caso de que la condición se cumpla, no sucede nada: el programa continúa con su flujo normal. Esto es indicativo de que las expectativas que teníamos se han satisfecho.

Sin embargo, si la condición que fijamos no se cumpla, la aserción devuelve un error `AssertionError` y el programa interrupme su ejecución:

```
>>> result = -1

>>> assert result > 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Podemos observar que la excepción que se lanza no contiene ningún mensaje informativo. Es posible personalizar este mensaje añadiendo un segundo elemento en la *tupla* de la aserción:

```
>>> assert result > 0, 'El resultado debe ser positivo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: El resultado debe ser positivo
```

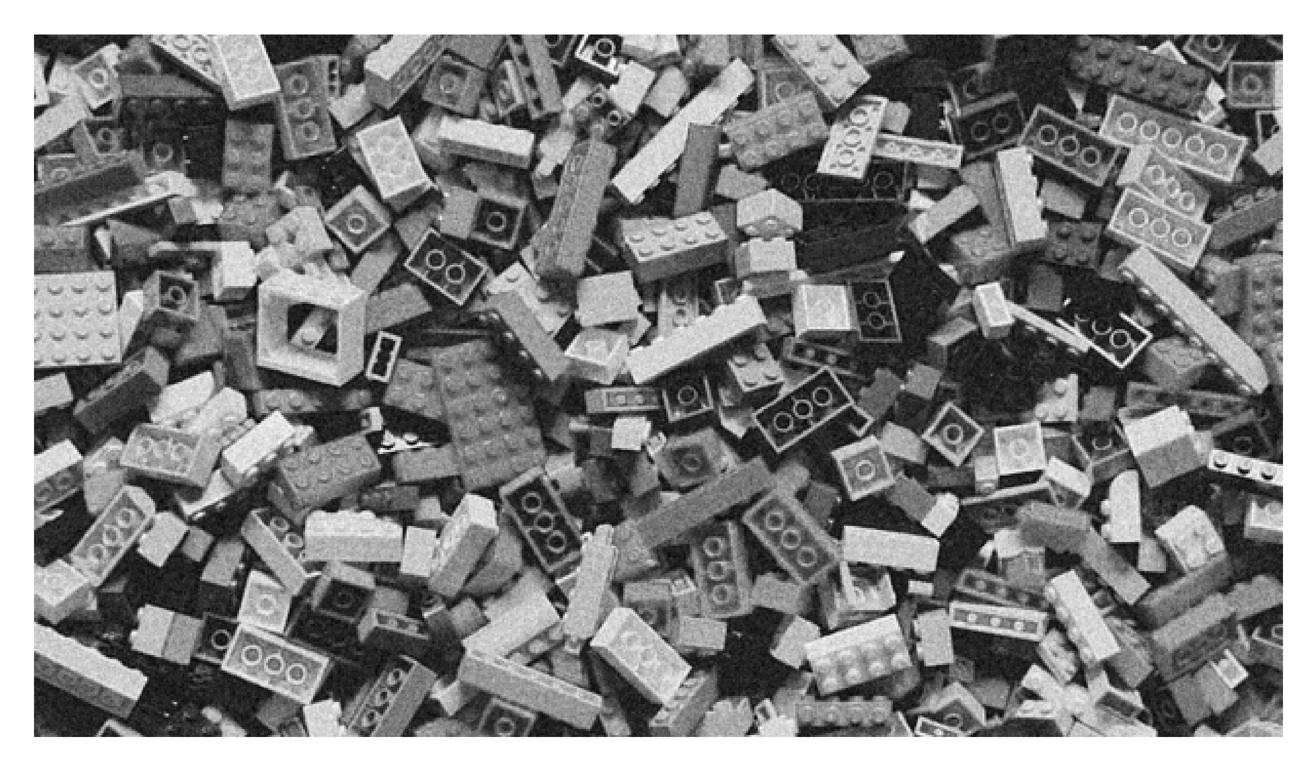
## EJERCICIOS DE REPASO

1. Escriba una clase `Card` que represente una carta de poker y una clase `InvalidCardError` que represente un error propio indicando que la carta no es válida.

## AMPLIAR CONOCIMIENTOS

- Python Exceptions: An introduction
- Python KeyError Exceptions and How to Handle Them
- Understanding the Python Traceback

## 6.4 Módulos



Escribir pequeños trozos de código puede resultar interesante para realizar determinadas pruebas o funcionalidades sencillas. Pero a la larga, nuestros programas tenderán a crecer y será necesario agrupar el código en piezas más manejables.[1]

### 6.4.1 Organización

Los **módulos** son simplemente ficheros de texto que contienen código Python y representan unidades con las que *evitar la repetición* y *favorecer la reutilización*.

Los módulos pueden agruparse en carpetas denominadas **paquetes** mientras que estas carpetas, a su vez, pueden dar lugar a **librerías**.

Un ejemplo de todo ello lo encontramos en la librería estándar. Se trata de una librería que ya viene incorporada en Python y que, a su vez, dispone de una serie de paquetes que incluyen distintos módulos.

Un caso concreto dentro de la `stdlib` (libería estándar) podría ser el del paquete `urllib` – para operaciones con URLs – que dispone de 5 módulos:

[1] Foto original por Xavi Cabrera en Unsplash.

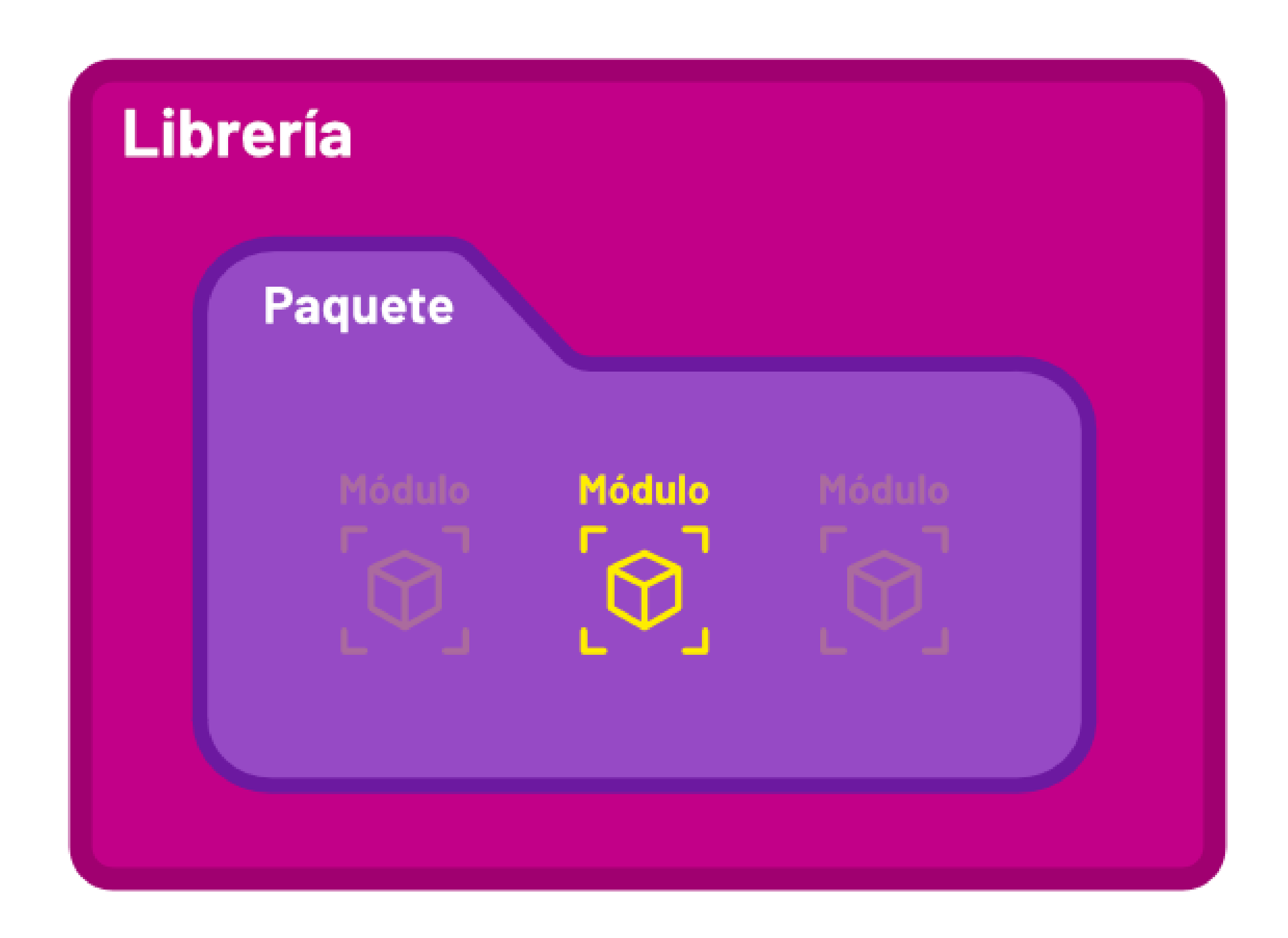



Figura 21: Concepto de módulo, paquete y librería

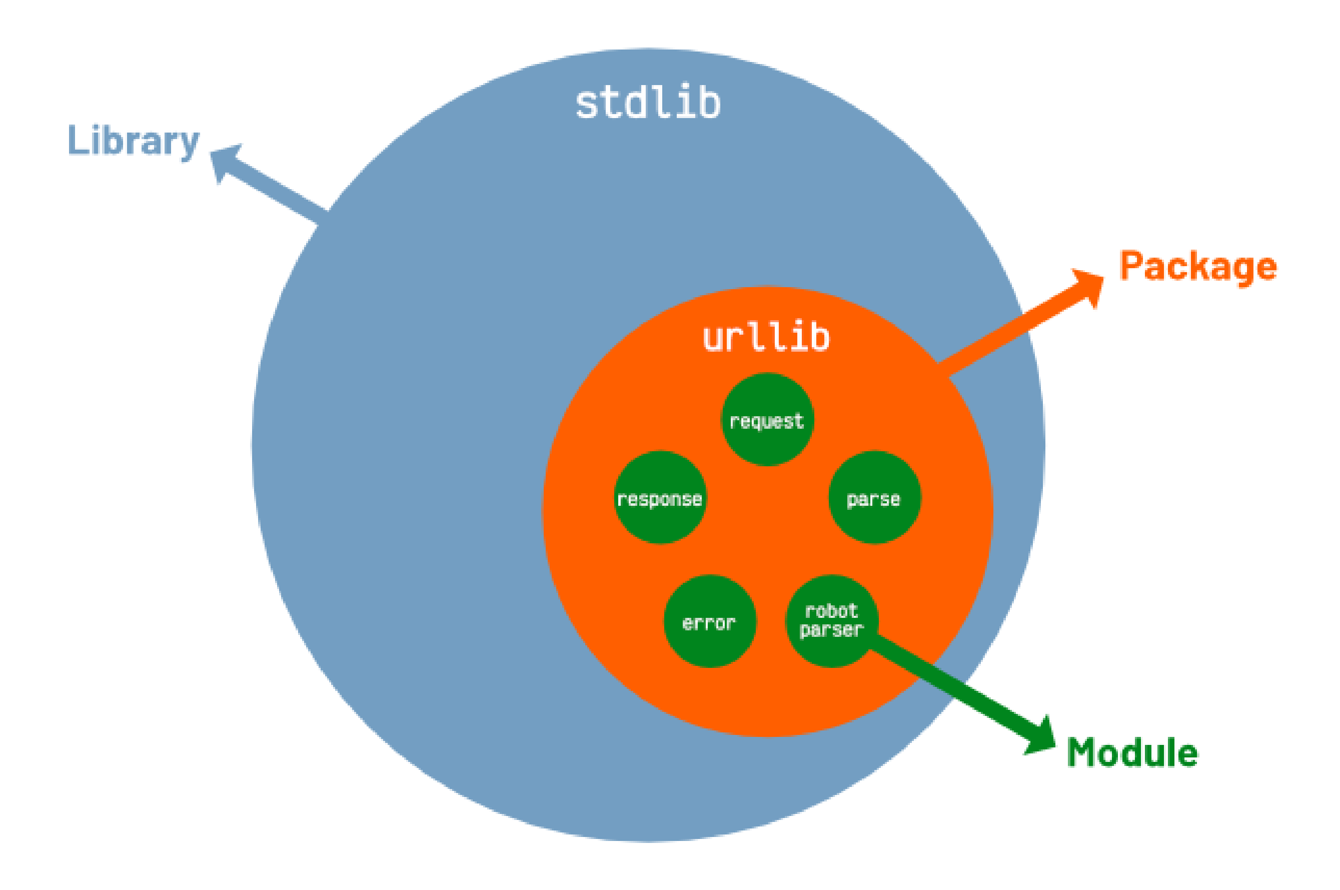



Figura 22: Ejemplo de paquete y módulos en la librería estándar

### 6.4.2 Importar un módulo

Para hacer uso del código de otros módulos usaremos la sentencia `import`. Esto permite importar el código y las variables de dicho módulo para tenerlas disponibles en nuestro programa.

La forma más sencilla de importar un módulo es `import <module>` donde `module` es el nombre de otro fichero Python, sin la extensión `.py`.

Supongamos que partimos del siguiente fichero (*módulo*) `stats.py`:

```python
def mean(*values: int | float) -> float:
    '''Calculate mean of values'''
    return sum(values) / len(values)


def std(*values: int | float) -> float:
    '''Calculate standard deviation of values'''
    m = mean(*values)
    p = sum((v - m) ** 2 for v in values)
    return (p / (len(values) - 1)) ** (1 / 2)
```

Desde otro fichero podríamos hacer uso de las funciones definidas en `stats.py`.

#### Importar módulo completo

Desde otro fichero (en principio en la misma carpeta) haríamos lo siguiente para importar todo el contenido del módulo `stats.py`:

```python
1 >>> import stats
2
3 >>> stats.mean(6, 3, 9, 5)
4 5.75
5 >>> stats.std(6, 3, 9, 5)
6 2.5
```

**Nota:** Nótese que en la **línea 3** debemos anteponer a la función `mean()` el *espacio de nombres* que define el módulo `stats`.

En el caso de utilizar un **módulo de la librería estándar**, basta con saber su nombre e importarlo directamente:

```python
>>> import os
```

### Ruta de búsqueda de módulos

Cuando importamos un módulo en Python el intérprete trata de encontrarlo (por orden) en las rutas definidas en la variable `sys.path`. Veamos su contenido (para mi caso concreto):

```
>>> import sys

>>> sys.path
['/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/bin',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python39.zip',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9/lib-dynload',
 '',
 '/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-
↪packages']
```

**Importante:** La **cadena vacía** `''` en la lista `sys.path` hace referencia a la **carpeta actual** de trabajo.

### Modificando la ruta de búsqueda

Si queremos modificar la ruta de búsqueda, existen dos opciones:

**Modificando directamente la variable `PYTHONPATH`** Para ello exportamos dicha variable de entorno desde una terminal:

```
$ export PYTHONPATH=/tmp
```

Y comprobamos que se ha modificado en `sys.path`:

```
>>> sys.path
['/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/bin',
 '/tmp',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python39.zip',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9/lib-dynload',
 '',
 '/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-
↪packages']
```

**Modificando directamente la lista `sys.path`** Para ello accedemos a lista que está en el módulo `sys` de la librería estandar:

```
>>> sys.path.append('/tmp')  # añadimos al final

>>> sys.path
['/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/bin',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python39.zip',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9/lib-dynload',
 '',
 '/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-
↪packages',
 '/tmp']
```

```
>>> sys.path.insert(0, '/tmp')  # insertamos por el principio

>>> sys.path
['/tmp',
 '/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/bin',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python39.zip',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9',
 '/Users/sdelquin/.pyenv/versions/3.9.1/lib/python3.9/lib-dynload',
 '',
 '/Users/sdelquin/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-
↪packages']
```

**Truco:** El hecho de poner nuestra ruta al principio o al final de `sys.path` influye en la búsqueda, ya que si existen dos (o más módulos) que se llaman igual en nuestra ruta de búsqueda, Python usará el primero que encuentre.

### Importar partes de un módulo

Es posible que no necesitemos todo aquello que está definido en `stats.py`. Supongamos que sólo vamos a calcular medias. Para ello haremos lo siguiente:

```
1 >>> from stats import mean
2
3 >>> mean(6, 3, 9, 5)
4 5.75
```

**Nota:** Nótese que en la **línea 3** ya podemos hacer uso directamente de la función `mean()` porque la hemos importado directamente. Este esquema tiene el inconveniente de la posible **colisión de nombres**, en aquellos casos en los que tuviéramos algún objeto con el mismo

nombre que el objeto que estamos importando.

---

Para **importar varios objetos** (funciones en este caso) desde un mismo módulo, podemos especificarlos separados por comas en la misma línea:

```
>>> from stats import mean, std
```

Es posible hacer `from stats import *` pero estaríamos importando todos los componentes del módulo, cuando a lo mejor no es lo que necesitamos. A continuación una imagen que define bien este escenario:




Figura 23: Diferencia entre importar un módulo o su contenido[2]

### Importar usando un alias

Hay ocasiones en las que interesa, por colisión de otros nombres o por mejorar la legibilidad, usar un nombre diferente del módulo (u objeto) que estamos importando. Python nos ofrece esta posibilidad a través de la sentencia `as`.

Supongamos que queremos importar la función del ejemplo anterior pero con otro nombre:

```
>>> from stats import mean as avg

>>> avg(6, 3, 9, 5)
5.75
```

[2] Imagen de ProgrammerHumor en Reddit.

### 6.4.3 Paquetes

Un **paquete** es simplemente una **carpeta** que contiene ficheros `.py`. Además permite tener una jerarquía con más de un nivel de subcarpetas anidadas.

Para ejemplificar este modelo vamos a crear un paquete llamado `extramath` que contendrá 2 módulos:

- `stats.py` para cálculos estadísticos.
- `frac.py` para operaciones auxiliares de fracciones.

El código del módulo de operaciones auxiliares de fracciones `frac.py` es el siguiente:

```python
def gcd(a: int, b: int) -> int:
    '''Greatest common divisor through Euclides Algorithm'''
    while b > 0:
        a, b = b, a % b
    return a


def lcm(a: int, b: int) -> int:
    '''Least common multiple through Euclides Algorithm'''
    return a * b // gcd(a, b)
```

Si nuestro código principal va a estar en un fichero `main.py` (*a primer nivel*), la estructura de ficheros nos quedaría tal que así:

```
1 .
2 □□□ main.py
3 □□□ extramath
4     □□□ frac.py
5     □□□ stats.py
6
7 1 directory, 3 files
```

**Línea 2** Punto de entrada de nuestro programa a partir del fichero `main.py`

**Línea 3** Carpeta que define el paquete `extramath`.

**Línea 4** Módulo para operaciones auxiliares de fracciones.

**Línea 5** Módulo para cálculos estadísticos.

### Importar desde un paquete

Si ya estamos en el fichero `main.py` (o a ese nivel) podremos hacer uso de nuestro paquete de la siguiente forma:

```
1 >>> from extramath import frac, stats
2
3 >>> frac.gcd(21, 35)
4 7
5
6 >>> stats.mean(6, 3, 9, 5)
7 5.75
```

**Línea 1** Importar los módulos `frac` y `stats` del paquete `extramath`

**Línea 3** Uso de la función `gcd` que está definida en el módulo `frac`

**Línea 5** Uso de la función `mean` que está definida en el módulo `stats`

## 6.4.4 Programa principal

Cuando decidimos desarrollar una pieza de software en Python, normalmente usamos distintos ficheros para ello. Algunos de esos ficheros se convertirán en *módulos*, otros se englobarán en *paquetes* y existirá uno en concreto que será nuestro **punto de entrada**, también llamado **programa principal**.

---

**Consejo:** Suele ser una buena práctica llamar `main.py` al fichero que contiene nuestro programa principal.

---

La estructura que suele tener este *programa principal* es la siguiente:

```
# imports de la librería estándar
# imports de librerías de terceros
# imports de módulos propios

# CÓDIGO PROPIO
# ...
# CÓDIGO PROPIO

if __name__ == '__main__':
    # punto de entrada real
```

Si queremos ejecutar este fichero `main.py` desde línea de comandos, tendríamos que hacer:

```
$ python main.py
```

---

**Nota:** Para llamar a función no es necesario que esté definida «antes» en el código, puede estar después. Pero siempre se trata de escribir el código intentado poner primero aquello que vamos a usar después.

---

### `if __name__ == '__main__'`

Esta condición permite, en el programa principal, diferenciar qué codigo se lanzará cuando el fichero se ejecuta directamente o cuando el fichero se importa desde otro lugar.

La variable `__name__` toma los siguientes valores:

- El nombre del módulo (o paquete) al **importar** el fichero.
- El valor `'__main__'` al **ejecutar** el fichero.

Supongamos el siguiente programa `hello.py` y analizemos cuál es el comportamiento según el escenario escogido:

```
1  import blabla
2
3
4  def myfunc():
5      print('Inside myfunc')
6      blabla.hi()
7
8
9  if __name__ == '__main__':
10     print('Entry point')
11     myfunc()
```

**`import hello`** El código se ejecuta siempre desde la primera instrucción a la última:

- **Línea 1**: se importa el módulo `blabla`.
- **Línea 4**: se define la función `myfunc()` y estará disponible para usarse.
- **Línea 9**: esta condición **no** se cumple, ya que estamos importando y la variable especial `__name__` no toma ese valor. Con lo cual finaliza la ejecución.
- *No hay salida por pantalla.*

**`$ python hello.py`** El código se ejecuta siempre desde la primera instrucción a la última:

- **Línea 1**: se importa el módulo `blabla`.
- **Línea 4**: se define la función `myfunc()` y estará disponible para usarse.

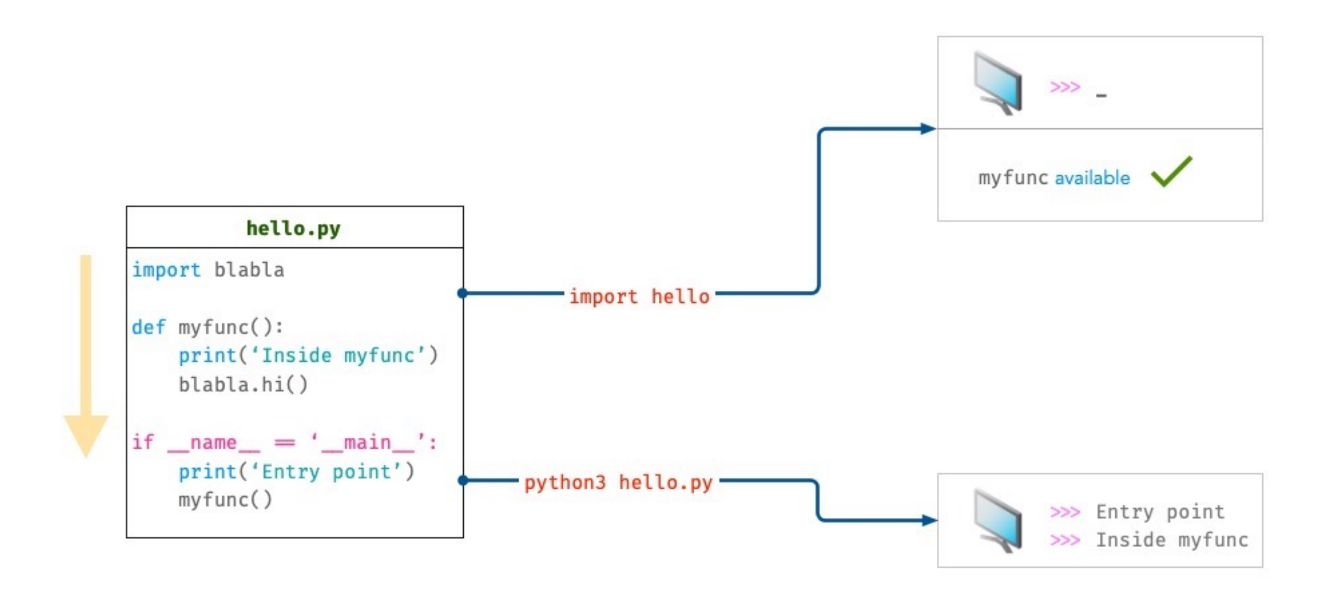



Figura 24: Comportamiento de un programa principal al importarlo o ejecutarlo

- **Línea 9**: esta condición **sí** se cumple, ya que estamos ejecutando directamente el fichero (*como programa principal*) y la variable especial `__name__` toma el valor `__main__`.
- **Línea 10**: salida por pantalla de la cadena de texto `Entry point`.
- **Línea 11**: llamada a la función `myfunc()` que muestra por pantalla `Inside myfunc`, además de invocar a la función `hi()` del módulo `blabla`.

## AMPLIAR CONOCIMIENTOS

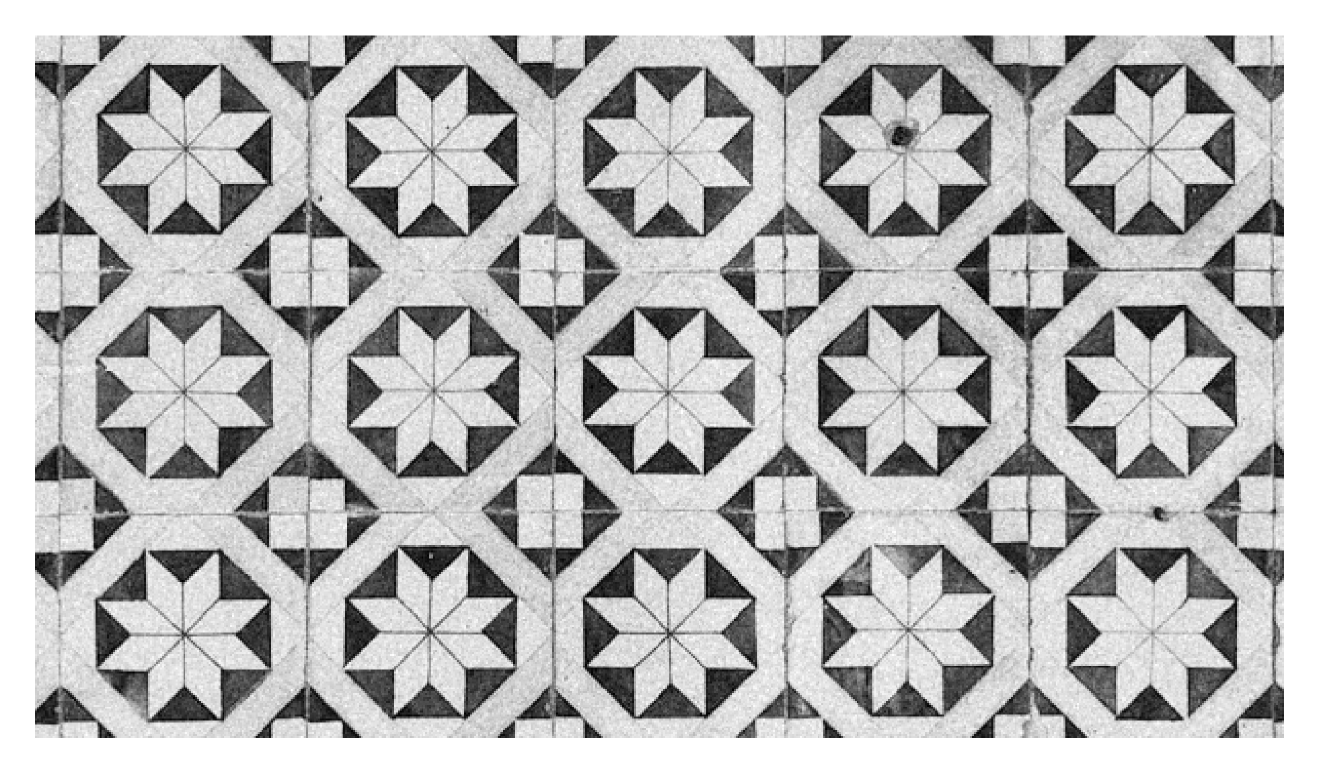
- Defining Main Functions in Python
- Python Modules and Packages: An Introduction
- Absolute vs Relative Imports in Python
- Running Python Scripts
- Writing Beautiful Pythonic Code With PEP 8
- Python Imports 101
- Clean Code in Python

# CAPÍTULO 7

# Procesamiento de texto

Además de las herramientas que se han visto en *cadenas de texto*, la librería estándar nos ofrece una serie de módulos para procesamiento de texto que nos harán la vida más fácil a la hora de gestionar este tipo de datos.

## 7.1 re



El módulo re permite trabajar con expresiones regulares.[1]

**Consejo:** Si tienes un problema y lo intentas resolver con expresiones regulares, entonces tienes dos problemas 😅 – Anónimo (Nótese la ironía)

### 7.1.1 ¿Qué es una expresión regular?

Una **expresión regular** (también conocida como **regex** o **regexp** por su contracción anglosajona **reg**-ular **exp**-ression) es una cadena de texto que conforma un **patrón de búsqueda**. Se utilizan principalmente para la *búsqueda de patrones* en cadenas de caracteres u *operaciones de sustituciones*.

Se trata de una herramienta ampliamente utilizada en las ciencias de la computación y necesaria para multitud de aplicaciones que traten con información textual.

Pero… ¿qué pinta tiene una expresión regular?

```
>>> regex = '^\d{8}[A-Z]$'
```

[1] Foto original de portada por Alice Butenko en Unsplash.

La expresión regular anterior nos permite comprobar que una cadena de texto dada es un DNI válido. Si analizamos parte por parte tendríamos lo siguiente:

- `^` comienzo de línea.
- `\d{8}` dígito que se repite 8 veces.
- `[A-Z]` letra en mayúsculas.
- `$` final de línea.

### 7.1.2 Sintaxis

Las expresiones regulares pueden contener tanto **caracteres especiales** como caracteres ordinarios. La mayoría de los caracteres ordinarios, como `'A'`, `'b'`, o `'0'` son las expresiones regulares más sencillas; simplemente se ajustan a sí mismas.

#### Caracteres especiales

Existen una serie de caracteres que tienen un significado especial dentro de una expresión regular:

| Caracter | Descripción |
| --- | --- |
| `.` | Coincide con cualquier carácter excepto con una nueva línea |
| `^` | Coincide con el comienzo de la cadena/línea |
| `$` | Coincide con el final de la cadena/línea |
| `*` | Coincide con 0 o más repeticiones de la expresión regular precedente |
| `+` | Coincide con 1 o más repeticiones de la expresión regular precedente |
| `?` | Coincide con 0 o 1 repetición de la expresión regular precedente |
| `{m}` | Coincide con exactamente `m` copias de la expresión regular precedente |
| `{m,n}` | Coincide de `m` a `n` copias de la expresión regular precedente, tratando de coincidir con el mayor número de repeticiones posibles. Omitiendo `m` se especifica un límite inferior de cero, y omitiendo `n` se especifica un límite superior infinito |
| `\` | Permite "escapar" el caracter que le sigue. Es decir, quitarle el significado especial que tiene |
| `[]` | Coincide con el conjunto de caracteres indicados dentro de los corchetes. Existe la variante `[a-f]` que coincide con el rango de caracteres entre la `a` y la `f`. Y también existe la variante `[^abc]` que coincide con todos los caracteres salvo los indicados dentro del conjunto. |
| `\|` | Coincide con una expresión regular u otra, separadas por este símbolo |

continué en la próxima página

Tabla 1 – proviene de la página anterior

| Caracter | Descripción |
|---|---|
| (...) | Coincide con cualquier expresión regular que esté dentro de los paréntesis, e indica el comienzo y el final de un grupo; el contenido de un grupo puede ser recuperado después de que se haya realizado una coincidencia |
| (?P<name>...) | Coincide con cualquier expresión regular que esté dentro de los paréntesis; el contenido del grupo es accesible por name |
| \number | Coincide con el contenido del grupo del mismo número. Se usa en conjunción con (...) |
| \b | Coincide con el comienzo o el final de una palabra |
| \B | Coincide con cualquier caracter que no sea cmienzo o final de una palabra |
| \d | Coincide con cualquier dígito decimal. Equivalente a [0-9] |
| \D | Coincide con cualquier carácter que no sea un dígito decimal. Equivalente a [^0-9] |
| \s | Coincide con cualquier espacio en blanco. Equivalente a [ \t\n\r\f\v] |
| \S | Coincide con cualquier carácter que no sea un espacio en blanco. Equivalente a [^ \t\n\r\f\v] |
| \w | Coincide con cualquier carácter alfanumérico. Equivalente a [a-zA-Z0-9_] |
| \W | Coincide con cualquier carácter que no sea un carácter alfanumérico. Equivalente a [^a-zA-Z0-9_] |
| | |
| | |

### Expresiones en crudo

Cuando definimos una expresión regular es conveniente utilizar el *formato raw* en las cadenas de texto para que los caracteres especiales no pierdan su semántica.

Veamos un ejemplo con el tabulador:

```
>>> regex = '\t[abc]$'
>>> print(regex)
    [abc]$

>>> regex = r'\t[abc]$'  # formato crudo
>>> print(regex)
\t[abc]$
```

## 7.1.3 Operaciones

### Buscar

La búsqueda de patrones es una de las principales utilidades de las expresiones regulares.

Supongamos que queremos buscar un número de teléfono dentro de un texto. Para ello vamos a utilizar la función search():

```
>>> import re

>>> text = 'Estaré disponible en el +34755142009 el lunes por la tarde'

>>> regex = r'\+?\d{2}\d{9}'
>>> re.search(regex, text)
<re.Match object; span=(24, 36), match='+34755142009'>
```

Esta función devuelve un objeto de tipo *Match* en cuya representación podemos ver un campo `span` que nos indica el alcance de la concidencia:

```
>>> text[24:36]
'+34755142009'
```

En el ejemplo anterior estamos buscando un solo elemento. Imaginemos un caso en el que queremos buscar todas las cantidades de dinero que aparecen en un determinado texto. Para ello vamos a utilizar la función findall():

```
>>> text = 'El coste ascendió a 36€ más un 12% de impuestos para un total de 40€'

>>> re.findall(r'\d+€', text)
['36€', '40€']
```

Si utilizamos un **grupo de captura** (paréntesis) la función `findall()` sólo nos devolverá aquellas coincidencias del grupo de captura:

```
>>> re.findall(r'(\d+)€', text)
['36', '40']
```

En el caso de que queramos agrupar expresiones regulares con `findall()` sin que se capturen estos grupos debemos utilizar la sintaxis: `(?:...)`

**Atención:** La función `findall()` no devuelve un objeto `Match` sino que retorna una lista con las cadenas de texto coincidentes.

### Coincidencia

El tipo de objeto Match es el utilizado en este módulo para representar una coincidencia.

Retomando el ejemplo anterior de la búsqueda del teléfono, veamos qué podemos hacer con este tipo de objetos:

```
>>> text = 'Estaré disponible en el +34755142009 el lunes por la tarde'

>>> regex = r'\+?\d{2}\d{9}'
>>> m = re.search(regex, text)

>>> m
<re.Match object; span=(24, 36), match='+34755142009'>
```

Si queremos acceder al texto completo coincidente, tenemos dos alternativas equivalentes:

```
>>> m[0]
'+34755142009'

>>> m.group(0)
'+34755142009'
```

Podemos conocer dónde empieza y dónde acaba el texto coincidente de la siguiente manera:

```
>>> m.span()  # equivale a m.span(0)
(24, 36)
```

Incluso hay una manera de acceder a estos índices por separado:

```
>>> m.start()
24

>>> m.end()
36
```

Si hubiera algún **subgrupo de búsqueda** podríamos acceder con los índices subsiguientes. Para ejemplificar este comportamiento vamos a modificar ligeramente la expresión regular original y capturar también el prefijo y el propio número de teléfono:

```
>>> m = re.search(r'\+?(\d{2})(\d{9})', text)
```

Ahora podemos acceder a los grupos capturados de distintas maneras:

```
>>> m.groups()
('34', '755142009')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> m[0]
'+34755142009'
>>> m[1]
'34'
>>> m[2]
'755142009'

>>> m.group()  # equivale a m.group(0)
'+34755142009'
>>> m.group(1)
'34'
>>> m.group(2)
'755142009'
```

Igualmente podemos acceder a los índices de comienzo y fin de cada grupo capturado:

```
>>> m.span(0)  # equivale a m.span()
(24, 36)

>>> m.span(1)  # '34'
(25, 27)

>>> m.span(2)  # '755142009'
(27, 36)
```

Por tanto, se cumple lo siguiente:

```
>>> for group_id in range(len(m.groups()) + 1):
...     start, end = m.span(group_id)
...     print(text[start:end])
...
+34755142009
34
755142009
```

Ahora vamos a **añadir nombres** a los **grupos de captura** para poder explicar otras funcionalidades de este objeto `Match`:

```
>>> regex = r'\+?(?P<prefix>\d{2})(?P<number>\d{9})'
>>> m = re.search(regex, text)
```

Tras este código, todo lo anterior sigue funcionando igual:

```
>>> m.groups()
('34', '755142009')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> m[1]
'34'

>>> m[2]
'755142009'
```

La diferencia está en que ahora podemos acceder a los grupos de captura por su nombre:

```
>>> m.group('prefix')
'34'
>>> m['prefix']
'34'

>>> m.group('number')
'755142009'
>>> m['number']
'755142009'
```

Y también existe la posibilidad de obtener el diccionario completo con los grupos capturados:

```
>>> m.groupdict()
{'prefix': '34', 'number': '755142009'}
```

### Ignorar mayúsculas y minúsculas

Supongamos que debemos encontrar todas las vocales que hay en un determinado nombre. La primera aproximación sería la siguiente:

```
>>> name = 'Alan Turing'
>>> regex = r'[aeiou]'

>>> re.findall(regex, name)
['a', 'u', 'i']
```

Aparentemente está bien pero nos damos cuenta de que la primera `A` mayúscula no está entre los resultados.

Este módulo de expresiones regulares establece una serie de «flags» que podemos pasar a las distintas funciones para modificar su comportamiento. Uno de los más importantes es el que nos permite ignorar mayúsculas y minúsculas: `re.IGNORECASE`.

Veamos su aplicación con el ejemplo anterior:

```
>>> re.findall(regex, name, re.IGNORECASE)
['A', 'a', 'u', 'i']
```

Podemos «abreviar» esta constante de la siguiente manera:

```
>>> re.findall(regex, name, re.I)
['A', 'a', 'u', 'i']
```

### Separar

Otras de las operaciones más usadas con expresiones regulares es la separación o división de una cadena de texto mediante un separador.

En su momento vimos el uso de la función *split()* para cadenas de texto, pero era muy limitada al especificar patrones avanzados. Veamos el uso de la función `re.split()` dentro de este módulo de expresiones regulares.

Un ejemplo muy sencillo sería **separar la parte entera de la parte decimal** en un determinado número flotante:

```
>>> regex = r'[.,]'

>>> re.split(regex, '3.14')
['3', '14']

>>> re.split(regex, '3,14')
['3', '14']
```

Vemos que la función devuelve una lista con los distintos elementos separados.

> **Prudencia:** Aunque parezca muy sencillo, este ejemplo no se puede resolver de manera «directa» usando la función `split()` de cadenas de texto.

### Reemplazar

Este módulo de expresions regulares también nos ofrece la posibilidad de reemplazar ocurrencias dentro de un texto.

A vueltas con el ejemplo del nombre de una persona, supongamos que recibimos la información en formato `<nombre> <apellidos>` y que la necesitamos en formato `<apellidos>, <nombre>`. Veamos cómo resolver este problema con la operación de reemplazar:

```
>>> name = 'Alan Turing'

>>> regex = r'(\w+) +(\w+)'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> repl = r'\2, \1'

>>> re.sub(regex, repl, name)
'Turing, Alan'
```

Hemos utilizado la función `re.sub()` que recibe 3 parámetros:

1. La expresión regular a localizar.
2. La expresión de reemplazo.
3. La cadena de texto sobre la que trabajar.

Dado que hemos utilizado *grupos de captura* podemos hacer referencia a ellos a través de sus índices mediante `\1`, `\2` y así sucesivamente.

Al igual que veíamos previamente, existe la posibilidad de nombrar los grupos de captura, y así facilitar la escritura de las expresiones de reemplazo:

```
>>> name = 'Alan Turing'

>>> regex = r'(?P<name>\w+) +(?P<surname>\w+)'

>>> repl = r'\g<surname>, \g<name>'

>>> re.sub(regex, repl, name)
'Turing, Alan'
```

Esta función admite un uso más avanzado ya que podemos **pasar una función** en vez de una cadena de texto de reemplazo, lo que nos abre un mayor rango de posibilidades.

Siguiendo con el caso anterior, supongamos que queremos hacer la misma transformación pero convirtiendo el apellido a mayúsculas, y asegurarnos de que el nombre queda como título:

```
>>> name = 'Alan Turing'

>>> regex = r'(\w+) +(\w+)'

>>> re.sub(regex, lambda m: f'{m[2].upper()}, {m[1].title()}', name)
'TURING, Alan'
```

**Ver también:**

Existe una función `re.subn()` que devuelve una tupla con la nueva cadena de texto reemplazada y el número de sustituciones realizadas.

### Casar

Si lo que estamos buscando es ver si una determinada cadena de texto «casa» (coincide) con un patrón de expresión regular, podemos hacer uso de la función `re.fullmatch()`.

Veamos un ejemplo en el que comprobamos si un texto dado es un DNI válido:

```
>>> regex = r'\d{8}[A-Z]'

>>> text = '54632178Y'

>>> re.fullmatch(regex, text)  # devuelve un objeto Match
<re.Match object; span=(0, 9), match='54632178Y'>
```

Si el patrón no casa la función devuelve `None`:

```
>>> text = '87896532$'

>>> re.fullmatch(regex, text)  # devuelve None

>>> re.fullmatch(regex, text) is None
True
```

Todo esto lo podemos poner dentro una sentencia condicional haciendo uso además del *operador morsa* para aprovechar la variable creada:

```
>>> def check_id_card(id_card: text) -> None:
...     if m := re.fullmatch(regex, id_card):
...         print(f'{text} es un DNI válido')
...         print(m.span())
...     else:
...         print(f'{text} no es un DNI válido')
...

>>> check_id_card('54632178Y')
54632178Y es un DNI válido
(0, 9)

>>> check_id_card('87896532$')
87896532$ no es un DNI válido
```

Hay una **variante más «flexible»** para casar que es `re.match()` y comprueba la existencia del patrón **sólo desde el comienzo de la cadena**. *Es decir, que si el final de la cadena no coincide sigue casando.*

Continuando con el caso anterior de comprobación de los DNI, podemos ver que añadir caracteres al final del documento de identidad no modifica el comportamiento de `re.match()`:

```
>>> regex = r'\d{8}[A-Z]'
>>> text = '54632178Y###'

>>> re.match(regex, text)
<re.Match object; span=(0, 9), match='54632178Y'>
```

Sin embargo no sucede lo mismo si añadimos caracteres al principio de la cadena:

```
>>> regex = r'\d{8}[A-Z]'
>>> text = '&&&54632178Y###'

>>> re.match(regex, text)
# None!
```

En cualquier caso podemos hacer que `re.match()` se comporte como `re.fullmatch()` si especificamos los **indicadores de comienzo y final de línea** en el patrón:

```
>>> regex = r'^\d{8}[A-Z]$'
>>> text = '54632178Y'

>>> re.match(regex, text)
<re.Match object; span=(0, 9), match='54632178Y'>
```

**Truco:** Tanto `re.fullmatch()` como `re.match()` devuelven un objeto de tipo *Match* con lo que podemos hacer uso de todos sus métodos y atributos.

### Compilar

Si vamos a utilizar una expresión regular una única vez entonces no debemos preocuparnos por cuestiones de rendimiento. Pero si repetimos su aplicación, sería más recomendable **compilar** la expresión regular a un patrón para mejorar el rendimiento:

```
>>> regex = r'\d+'

>>> r = re.compile(regex)

>>> type(r)
re.Pattern

>>> re.search(r, '1:abc;10:def;100;ghi')
<re.Match object; span=(0, 1), match='1'>
```

### EJERCICIOS DE REPASO

1. Escriba un programa en Python que encuentre todas las palabras que comiencen por vocal en un texto dado: `vowel_words.py`
2. Escriba un programa en Python que indique si una URL dada es válida o no: `valid_url.py`
3. Escriba un programa en Python que indique si un determinado número es o no un *flotante válido en Python*: `valid_float.py`
4. Escriba un programa en Python que determine si un email dado tiene el formato correcto: `valid_email.py`
5. Escriba un programa en Python que obtenga el resultado de una operación entre números enteros positivos. Las operación puede ser suma, resta, multiplicación o división, y puede haber espacios (o no) entre los operandos y el operador: `eval_str_op.py`

## 7.2 string



El módulo string proporciona una serie de constantes muy útiles para manejo de «strings», además de distintas estrategias de formateado de cadenas.[1]

[1] Foto original de portada por Steve Johnson en Unsplash.

### 7.2.1 Constantes

Las constantes definidas en este módulo son las siguientes:

```
>>> import string

>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'

>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> string.digits
'0123456789'

>>> string.hexdigits
'0123456789abcdefABCDEF'

>>> string.octdigits
'01234567'

>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?
↪@[\\]^_`{|}~ \t\n\r\x0b\x0c'

>>> string.whitespace
' \t\n\r\x0b\x0c'
```

**Ejercicio**

Dada una cadena de texto, compruebe si todos sus caracteres son dígitos ASCII. *Ignore los espacios en blanco.*

**Ejemplo**

- Entrada: `This is it`
- Salida: True

### 7.2.2 Plantillas

El módulo `string` también nos permite usar plantillas con interpolación de variables. Algo similar a los *f-strings* pero con otro tipo de sintaxis.

Lo primero es definir la plantilla. Las variables que queramos interporlar deben ir precedidas del signo dólar `$`:

```
from string import Template

tmpl = Template('$lang is the best programming language in the $place!')
```

Ahora podemos realizar la sustitución con los valores que nos interesen:

```
>>> tmpl.substitute(lang='Python', place='World')
'Python is the best programming language in the World!'

>>> tmpl.substitute({'lang': 'Python', 'place': 'World'})
'Python is the best programming language in the World!'
```

Hay que prestar atención cuando el identificador de variable está seguido por algún carácter que, a su vez, puede formar parte del identificador. En este caso hay que utilizar llaves para evitar la ambigüedad:

```
>>> tmpl = Template('You won several ${price}s')

>>> tmpl.substitute(price='phone')
'You won several phones'
```

#### Sustitución segura

En el caso de que alguna de las variables que estamos interpolando no exista o no tenga ningún valor, obtendremos un error al sustituir:

```
>>> tmpl = Template('$lang is the best programming language in the $place!')

>>> tmpl.substitute(lang='Python')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'place'
```

Para ello Python nos ofrece el método `safe_substitute()` que no emite error si alguna variable no es especificada:

```
>>> tmpl.safe_substitute(lang='Python')
'Python is the best programming language in the $place!'
```

### Casos de uso

A primera vista podría parecer que este sistema de plantillas no aporta gran ventaja sobre los *f-strings* que ya hemos visto. Sin embargo hay ocasiones en los que puede resultar muy útil.

La mayoría de estas situaciones tienen que ver con **la oportunidad** de definir el «string». Si en el momento de crear la plantilla aún no están disponibles las variables de sustitución, podría interesar utilizar la estrategia que nos proporciona este módulo.

Supongamos un ejemplo en el que tenemos una estructura de «url» y queremos únicamente sustituir una parte de ella. Para no tener que repetir la cadena de texto completa en un «f-string», podríamos seguir este enfoque:

```
>>> urlbase = Template('https://python.org/3/library/$module.html')

>>> for module in ('string', 're', 'difflib'):
...     url = urlbase.substitute(module=module)
...     print(url)
...
https://python.org/3/library/string.html
https://python.org/3/library/re.html
https://python.org/3/library/difflib.html
```

CAPÍTULO 8

# Acceso a datos

Además de las herramientas que se han visto en *cadenas de texto*, la librería estándar nos ofrece una serie de módulos para procesamiento de texto que nos harán la vida más fácil a la hora de gestionar este tipo de datos.

## 8.1 sqlite



El módulo sqlite3 permite trabajar con bases de datos de tipo **SQLite**[1].

### 8.1.1 ¿Qué es SQLite?

SQLite es un sistema gestor de bases de datos relacional contenido en una pequeña librería escrita en C (~275kB).

A continuación se muestran algunas de sus **principales características**:

- Tablas, índices, «triggers» y vistas ilimitadas.
- Hasta 32K columnas en una tabla y filas ilimitadas.
- Índices multi-columna.
- Restricciones de tipo `CHECK`, `UNIQUE`, `NOT NULL` y `FOREIGN KEY`.
- Transacciones planas usando `BEGIN`, `COMMIT` y `ROLLBACK`
- Transacciones anidadas usando `SAVEPOINT`, `RELEASE` y `ROLLBACK TO`.
- Subconsultas.
- «Joins» de hasta 64 relaciones.

[1] Foto original de portada por Jandira Sonnendeck en Unsplash.

- «Joins» de tipo «left», «right» y «full outer».
- Uso de `DISTINCT`, `ORDER BY`, `GROUP BY`, `HAVING`, `LIMIT` y `OFFSET`.
- Uso de `UNION`, `UNION ALL`, `INTERSECT` y `EXCEPT`.
- Una amplia librería de funciones SQL estándar.
- Funciones de agregación.
- Funciones de ventana.
- Por supuesto el uso de `UPDATE`, `DELETE` e `INSERT`.
- Cláusula UPSERT.
- Soporte para valores JSON.

Y muchas otras que se pueden consultar en la página del proyecto.

### 8.1.2 Conexión a la base de datos

Una base de datos SQLite no es más que un fichero binario, habitualmente con extensión `.db` o `.sqlite`. Antes de realizar cualquier operación es necesario «conectar» con este fichero.

La **conexión a la base de datos** se realiza a través de la función connect() que espera recibir la ruta al fichero de base de datos y devuelve un objeto de tipo Connection:

```
>>> import sqlite3

>>> con = sqlite3.connect('python.db')

>>> con
<sqlite3.Connection at 0x106ea8210>
```

> **Advertencia:** El módulo se llama **sqlite3** (no olvidarse del 3 al final).

La función `connect()` creará el fichero `python.db` (si es que no existe ya). En un principio no tiene contenido alguno:

```
>>> !file python.db
python.db: empty
```

Una vez que disponemos de la conexión ya podemos obtener un Cursor mediante la función cursor(). Un **cursor** se podría ver como un manejador para realizar operaciones sobre la base de datos:

```
>>> cur = con.cursor()

>>> cur
<sqlite3.Cursor at 0x106a63960>
```

### 8.1.3 Creación de tablas

Para poder crear una tabla primero debemos manejar los tipos de datos SQLite disponibles. Aunque hay alguno más, con los siguientes nos será suficiente para la inmensa mayoría de diseños de bases de datos que podamos necesitar:

- `INTEGER` para valores enteros.
- `REAL` para valores flotantes.
- `TEXT` para cadenas de texto.

> **Prudencia:** Aunque `INT` también está permitido, se desaconseja su uso en favor de `INTEGER` especialmente cuando trabajamos con la librería Python `sqlite3` y no queremos obtener resultados inesperados.

Durante toda esta sección vamos a trabajar con una tabla de ejemplo que represente las distintas versiones de Python que han sido liberadas.

Empecemos creando la tabla `pyversions` a través de un código SQL similar al siguiente:

```
CREATE TABLE pyversions (
    branch TEXT PRIMARY KEY,
    released_at_year INTEGER,
    released_at_month INTEGER,
    release_manager TEXT
)
```

Haremos uso del cursor creado para **ejecutar** estas instrucciones:

```
>>> sql = '''CREATE TABLE pyversions (
...     branch TEXT PRIMARY KEY,
...     released_at_year INTEGER,
...     released_at_month INTEGER,
...     release_manager TEXT
... )'''

>>> cur.execute(sql)
<sqlite3.Cursor at 0x106a63960>
```

**Consejo:** Las *cadenas multilínea* son grandes aliadas a la hora de escribir sentencias SQL.

Ya hemos creado la tabla `pyversions` de manera satisfactoria.

Si comprobamos ahora el contenido del fichero `python.db` podemos observar que nos indica la versión de SQLite y la última escritura:

```
>>> !file python.db
python.db: SQLite 3.x database, last written using SQLite version 3032003
```

### 8.1.4 Añadiendo datos

Para tener contenido sobre el que trabajar, vamos primeramente a añadir ciertos datos a la tabla. Como básicamente seguimos ejecutando sentencias SQL (en este caso de inserción) podemos volver a hacer uso de la función execute():

```
>>> sql = 'INSERT INTO pyversions VALUES ("2.6", 2008, 10, "Barry Warsaw")'

>>> cur.execute(sql)
<sqlite3.Cursor at 0x106a63960>
```

Aparentemente todo ha ido bien. Vamos a usar – temporalmente – la herramienta cliente `sqlite3`[2] para ver el contenido de la tabla:

```
$ sqlite3 python.db "select * from pyversions"
$ # Vacío
```

Resulta que no obtenemos ningún registro. ¿Por qué ocurre esto? Se debe a que la transacción está aún pendiente de confirmar. Para consolidarla tendremos que hacer uso de la función commit():

```
>>> con.commit()
```

**Ver también:**

Cada vez que usamos la función `execute()` comienza una **nueva transacción** a la base de datos que debe confirmarse con `commit()` o bien deshacerse con rollback().

Ahora podemos comprobar que sí se han guardado los datos correctamente:

```
$ sqlite3 python.db "select * from pyversions"
2.6|2008|10|Barry Warsaw
```

[2] Herramienta cliente de sqlite para terminal.

**Nota:** La función `commit()` pertenece al objeto conexión, no al objeto cursor.

### Inserciones parametrizadas

Supongamos que no sabemos, a priori, los datos que vamos a insertar en la tabla puesto que provienen del usuario o de otra fuente externa. En este caso cabría plantearse cuál es la mejor opción para parametrizar la consulta.

#### Usando f-strings

Una primera aproximación podrían ser los *f-strings* a través de una simple interpolación de variables. Veamos un ejemplo de ello:

```
>>> branch = 3.9
>>> released_at_year = 2020
>>> released_at_month = 10
>>> release_manager = 'Łukasz Langa'

>>> sql = f'INSERT INTO pyversions VALUES ({branch}, {released_at_year}, {released_
↪at_month}, {release_manager})'
>>> sql
'INSERT INTO pyversions VALUES (3.9, 2020, 10, Łukasz Langa)'

>>> cur.execute(sql)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OperationalError: near "Langa": syntax error
```

Obtenemos un error porque el contenido de «release manager» **es una cadena de texto y no puede contener espacios**. Una solución a este problema sería detectar qué campos necesitan comillas e incorporarlas de forma manual.

#### Usando placeholders SQLite

Pero existe otra aproximación y es **usar los «placeholders» que ofrece SQLite** al ejecutar sentencias. Estos «placeholders» se representan por el **símbolo de interrogación** `?` y se sustituyen por el **valor correspondiente en una tupla (o iterable)** que pasamos como parámetro a posteriori.

Veamos cómo sería esta reimplementación:

```
>>> sql = 'INSERT INTO pyversions VALUES (?, ?, ?, ?)'

>>> cur.execute(sql, (branch, released_at_year, released_at_month, release_manager))
<sqlite3.Cursor at 0x107426c40>
```

Ahora sí que todo ha ido bien y **no nos hemos tenido que preocupar del tipo de los campos**. Ya sólo por esto valdría la pena utilizar esta aproximación pero también ayuda a evitar ataques por inyección SQL[4].

**Prudencia:** Cuando sólo haya un «placeholder» hay que recordar que las *tuplas de un único elemento* necesitan una coma al final: `cur.execute('INSERT INTO table (column) VALUES (?)', (value,))`

En estos casos quizás sea incluso más sencillo pasar una lista de un elemento: `[value]`

Este módulo nos ofrece igualmente la posibilidad de usar **parámetros nominales a través de un diccionario** especificando los campos con dos puntos `:field`. Veamos cómo sería esta aproximación:

```
>>> sql = 'INSERT INTO pyversions VALUES (:branch, :year, :month, :manager)'

>>> cur.execute(sql, dict(year=2020, month=10, branch=3.9, manager='Łukasz Langa'))
<sqlite3.Cursor at 0x107426c40>
```

**Truco:** Nótese que no es necesario usar el mismo orden de los parámetros cuando utilizamos esta aproximación nominal ya que el diccionario incluye las claves.

### Inserciones en lote

Vamos a pensar en un escenario algo más real, en el que necesitamos **insertar en la tabla más de un registro**. Obviamente la solución programática no puede ser ir de uno en uno.

Supongamos que disponemos del siguiente fichero `pyversions.csv`:

```
branch,year,month,manager
2.6,2008,10,Barry Warsaw
2.7,2010,7,Benjamin Peterson
3.0,2008,12,Barry Warsaw
```

(continué en la próxima página)

[4] Inyección SQL es un método de infiltración de código intruso que se vale de una vulnerabilidad informática presente en una aplicación en el nivel de validación de las entradas para realizar operaciones sobre una base de datos.

(proviene de la página anterior)

```
3.1,2009,6,Benjamin Peterson
3.2,2011,2,Georg Brandl
3.3,2012,9,Georg Brandl
3.4,2014,3,Larry Hastings
3.5,2015,9,Larry Hastings
3.6,2016,12,Ned Deily
3.7,2018,6,Ned Deily
3.8,2019,10,Łukasz Langa
3.9,2020,10,Łukasz Langa
3.10,2021,10,Pablo Galindo Salgado
3.11,2022,10,Pablo Galindo Salgado
3.12,2023,10,Thomas Wouters
```

Queremos procesar cada línea e insertarla en la tabla como un nuevo registro. Veamos una primera aproximación:

```
>>> with open('pyversions.csv') as f:
...     f.readline()  # ignore headers
...     for line in f:
...         branch, year, month, manager = line.strip().split(',')
...         sql = f'INSERT INTO pyversions VALUES ("{branch}", {year}, {month}, "
↪{manager}")'
...         cur.execute(sql)
...     con.commit()
...
```

Pero este módulo permite atacar el problema desde otro enfoque utilizando la función executemany(). Esta función admite un **iterable de iterables** (con el mismo número de campos que la tabla) desde donde recupera los datos:

```
>>> f = open('pyversions.csv')
>>> data = [line.strip().split(',') for line in f.readlines()[1:]]
>>> data
[['2.6', '2008', '10', 'Barry Warsaw'],
 ['2.7', '2010', '7', 'Benjamin Peterson'],
 ['3.0', '2008', '12', 'Barry Warsaw'],
 ['3.1', '2009', '6', 'Benjamin Peterson'],
 ['3.2', '2011', '2', 'Georg Brandl'],
 ['3.3', '2012', '9', 'Georg Brandl'],
 ['3.4', '2014', '3', 'Larry Hastings'],
 ['3.5', '2015', '9', 'Larry Hastings'],
 ['3.6', '2016', '12', 'Ned Deily'],
 ['3.7', '2018', '6', 'Ned Deily'],
 ['3.8', '2019', '10', 'Łukasz Langa'],
 ['3.9', '2020', '10', 'Łukasz Langa'],
```

(continué en la próxima página)

(proviene de la página anterior)

```
 ['3.10', '2021', '10', 'Pablo Galindo Salgado'],
 ['3.11', '2022', '10', 'Pablo Galindo Salgado'],
 ['3.12', '2023', '10', 'Thomas Wouters']]

>>> sql = 'INSERT INTO pyversions VALUES (?, ?, ?, ?)'
>>> cur.executemany(sql, data)
<sqlite3.Cursor at 0x104f3fb20>

>>> con.commit()
```

Si dispusiéramos de un **diccionario** podríamos indicar incluso el nombre de los campos:

```
>>> f = open('pyversions.csv')
>>> fields = f.readline().strip().split(',')
>>> data = [{f: v for f, v in zip(fields, line.strip().split(','))} for line in f]

>>> data
[{'branch': '2.6', 'year': '2008', 'month': '10', 'manager': 'Barry Warsaw'},
 {'branch': '2.7', 'year': '2010', 'month': '7', 'manager': 'Benjamin Peterson'},
 {'branch': '3.0', 'year': '2008', 'month': '12', 'manager': 'Barry Warsaw'},
 {'branch': '3.1', 'year': '2009', 'month': '6', 'manager': 'Benjamin Peterson'},
 {'branch': '3.2', 'year': '2011', 'month': '2', 'manager': 'Georg Brandl'},
 {'branch': '3.3', 'year': '2012', 'month': '9', 'manager': 'Georg Brandl'},
 {'branch': '3.4', 'year': '2014', 'month': '3', 'manager': 'Larry Hastings'},
 {'branch': '3.5', 'year': '2015', 'month': '9', 'manager': 'Larry Hastings'},
 {'branch': '3.6', 'year': '2016', 'month': '12', 'manager': 'Ned Deily'},
 {'branch': '3.7', 'year': '2018', 'month': '6', 'manager': 'Ned Deily'},
 {'branch': '3.8', 'year': '2019', 'month': '10', 'manager': 'Łukasz Langa'},
 {'branch': '3.9', 'year': '2020', 'month': '10', 'manager': 'Łukasz Langa'},
 {'branch': '3.10', 'year': '2021', 'month': '10', 'manager': 'Pablo Galindo Salgado
↪'},
 {'branch': '3.11', 'year': '2022', 'month': '10', 'manager': 'Pablo Galindo Salgado
↪'},
 {'branch': '3.12', 'year': '2023', 'month': '10', 'manager': 'Thomas Wouters'}]

>>> sql = 'INSERT INTO pyversions VALUES (:branch, :year, :month, :manager)'
>>> cur.executemany(sql, data)
<sqlite3.Cursor at 0x106e96030>

>>> con.commit()
```

En cualquiera de los tres casos anteriores el resultado es el mismo y los registros quedan correctamente insertados en la base de datos:

```
$ sqlite3 python.db "SELECT * FROM pyversions"
```

(continué en la próxima página)

(proviene de la página anterior)

```
2.6|2008|10|Barry Warsaw
2.7|2010|7|Benjamin Peterson
3.0|2008|12|Barry Warsaw
3.1|2009|6|Benjamin Peterson
3.2|2011|2|Georg Brandl
3.3|2012|9|Georg Brandl
3.4|2014|3|Larry Hastings
3.5|2015|9|Larry Hastings
3.6|2016|12|Ned Deily
3.7|2018|6|Ned Deily
3.8|2019|10|Łukasz Langa
3.9|2020|10|Łukasz Langa
3.10|2021|10|Pablo Galindo Salgado
3.11|2022|10|Pablo Galindo Salgado
3.12|2023|10|Thomas Wouters
```

### Identificador de fila

En el comportamiento por defecto de una base de datos SQLite **todas las tablas disponen de una columna «oculta» denominada** `rowid` o identificador de fila.

Esta columna se va rellenando **de forma automática con valores enteros únicos** y puede utilizarse como clave primaria de los registros.

Para poder visualizar (o utilizar) esta columna es necesario indicarlo explícitamente en la consulta:

```
$ sqlite3 python.db "SELECT rowid, * FROM pyversions"
1|2.6|2008|10|Barry Warsaw
2|2.7|2010|7|Benjamin Peterson
3|3.0|2008|12|Barry Warsaw
4|3.1|2009|6|Benjamin Peterson
5|3.2|2011|2|Georg Brandl
6|3.3|2012|9|Georg Brandl
7|3.4|2014|3|Larry Hastings
8|3.5|2015|9|Larry Hastings
9|3.6|2016|12|Ned Deily
10|3.7|2018|6|Ned Deily
11|3.8|2019|10|Łukasz Langa
12|3.9|2020|10|Łukasz Langa
13|3.10|2021|10|Pablo Galindo Salgado
14|3.11|2022|10|Pablo Galindo Salgado
15|3.12|2023|10|Thomas Wouters
```

### Cerrando la conexión

Al igual que ocurre con un fichero de texto, es necesario **cerrar la conexión abierta** para que se liberen los recursos asociados y se desbloquee la base de datos.

La forma más directa de hacer esto sería:

```
>>> con.close()
```

> **Atención:** Si hay alguna transacción pendiente, esta no será guardada al cerrar la conexión con la base de datos, si previamente no se consolidan los cambios.

### Gestor de contexto

En SQLite también es posible utilizar un *gestor de contexto* sobre la conexión, que funciona de la siguiente manera:

- Si todo ha ido bien ejecutará un «commit» al final del bloque.
- Si ha habido alguna excepción ejecutará un «rollback» para que todo quede como al principio y deshacer los posibles cambios efectuados.

Ejemplo en el que todo va bien:

```
>>> try:
...     with con:
...         cur.execute('INSERT INTO pyversions VALUES ("3.13", 2024, 10, "Thomas␣
↪Wouters")')
... except sqlite3.IntegrityError:
...     print('Error: Duplicated Python version')
...

>>> con.close()
```

Ejemplo donde se produce un error:

```
>>> try:
...     with con:
...         cur.execute('INSERT INTO pyversions VALUES ("3.12", 2023, 10, "Thomas␣
↪Wouters")')
... except sqlite3.IntegrityError:
...     print('Error: Duplicated Python version')
...
Error: Duplicated Python version

>>> con.close()
```

Nótese que en ambos casos **debemos cerrar la conexión**. Esto no se realiza de forma automática.

Es interesante conocer las distintas excepciones que pueden producirse al trabajar con este módulo a la hora del control de errores y de plantear posibles escenarios de mejora.

## 8.1.5 Consultas

La manera más sencilla de hacer una consulta es **utilizar un cursor**. Existen dos aproximaciones en el tratamiento de los resultados de la consulta:

1. Registros como tuplas.
2. Registros como filas.

### Registros como tuplas

Cuando ejecutamos una consulta **el resultado es un objeto iterable** que podemos recorrer de la misma manera que hemos hecho hasta ahora. Los datos nos vienen en forma de **tuplas**:

```
>>> for row in cur.execute('SELECT * FROM pyversions'):
...     print(row)
...
('2.6', 2008, 10, 'Barry Warsaw')
('2.7', 2010, 7, 'Benjamin Peterson')
('3.0', 2008, 12, 'Barry Warsaw')
('3.1', 2009, 6, 'Benjamin Peterson')
('3.2', 2011, 2, 'Georg Brandl')
('3.3', 2012, 9, 'Georg Brandl')
('3.4', 2014, 3, 'Larry Hastings')
('3.5', 2015, 9, 'Larry Hastings')
('3.6', 2016, 12, 'Ned Deily')
('3.7', 2018, 6, 'Ned Deily')
('3.8', 2019, 10, 'Łukasz Langa')
('3.9', 2020, 10, 'Łukasz Langa')
('3.10', 2021, 10, 'Pablo Galindo Salgado')
('3.11', 2022, 10, 'Pablo Galindo Salgado')
('3.12', 2023, 10, 'Thomas Wouters')
('3.13', 2024, 10, 'Thomas Wouters')
```

También tenemos la opción de utilizar las funciones fetchone() y fetchall() para obtener una o todas las filas de la consulta:

```
>>> res = cur.execute('SELECT * FROM pyversions')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> res.fetchone()
('2.6', 2008, 10, 'Barry Warsaw')

>>> res.fetchall()
[('2.7', 2010, 7, 'Benjamin Peterson'),
 ('3.0', 2008, 12, 'Barry Warsaw'),
 ('3.1', 2009, 6, 'Benjamin Peterson'),
 ('3.2', 2011, 2, 'Georg Brandl'),
 ('3.3', 2012, 9, 'Georg Brandl'),
 ('3.4', 2014, 3, 'Larry Hastings'),
 ('3.5', 2015, 9, 'Larry Hastings'),
 ('3.6', 2016, 12, 'Ned Deily'),
 ('3.7', 2018, 6, 'Ned Deily'),
 ('3.8', 2019, 10, 'Łukasz Langa'),
 ('3.9', 2020, 10, 'Łukasz Langa'),
 ('3.10', 2021, 10, 'Pablo Galindo Salgado'),
 ('3.11', 2022, 10, 'Pablo Galindo Salgado'),
 ('3.12', 2023, 10, 'Thomas Wouters'),
 ('3.13', 2024, 10, 'Thomas Wouters')]
```

**Prudencia:** Nótese que la llamada a `fetchone()` hace que quede «una fila menos» que recorrer. Es un comportamiento totalmente análogo a la *lectura de una línea* en un fichero.

### Registros como filas

Este módulo también nos permite obtener los resultados de una consulta como objetos de tipo Row lo que facilita acceder a los valores de cada registro **tanto por el índice como por el nombre de la columna**.

Para «activar» este modo tendremos que fijar el valor de la factoría de filas en la conexión:

```
>>> con = sqlite3.connect('python.db')
>>> con.row_factory = sqlite3.Row
```

**Importante:** Para que las consultas usen esta factoría hay que fijar el atributo `row_factory` **antes** de crear el cursor correspondiente.

Ahora creamos un cursor, ejecutamos la consulta y accedemos a la primera fila del resultado **como si fuera un diccionario**:

```
>>> cur = con.cursor()
>>> res = cur.execute('SELECT * FROM pyversions')

>>> row = res.fetchone()
>>> row
<sqlite3.Row at 0x107b76190>

>>> row.keys()
['branch', 'released_at_year', 'released_at_month', 'release_manager']

>>> row['branch']
'2.6'
>>> row['released_at_year']
2008
>>> row['released_at_month']
10
>>> row['release_manager']
'Barry Warsaw'
```

Pero también es posible seguir accediendo a la cada columna **a través del índice**:

```
>>> row[0]
'2.6'
>>> row[1]
2008
>>> row[2]
10
>>> row[3]
'Barry Warsaw'
```

### Desempaquetando filas

Cuando disponemos de una fila como resultado de una consulta (ya sea en formato tupla o en formato `sqlite3.Row`) podemos realizar un desempaquetado para separar sus campos en variables únicas:

```
>>> sql = 'SELECT * FROM pyversions'
>>> result = cur.execute(sql)
>>> row = result.fetchone()

>>> row
<sqlite3.Row at 0x102e71ab0>

>>> branch, released_at_year, released_at_month, release_manager = row
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> branch
'2.6'
>>> released_at_year
2008
>>> released_at_month
10
>>> release_manager
'Barry Warsaw'
```

### Número de filas

Hay ocasiones en las que lo que necesitamos obtener no es el dato en sí mismo, sino el **número de filas vinculadas a una determinada consulta**. En este sentido hay varias alternativas.

La primera aproximación es **utilizar herramientas Python** para obtener la longitud del resultado de la consulta:

```
>>> result = cur.execute('SELECT * FROM pyversions')

>>> rows = result.fetchall()

>>> len(rows)
15
```

La segunda aproximación es **mediante la sentencia SQL para contar**: `COUNT()` y obtener su resultado:

```
>>> result = cur.execute('SELECT COUNT(*) FROM pyversions')

>>> rows = result.fetchone()

>>> rows[0]  # sólo hay una columna
15
```

Obviamente si lo único que necesitamos es obtener el número de filas afectadas, esta segunda opción a través de `COUNT()` tiene más sentido.

### Comprobando si hay resultados

Hay ocasiones en las que necesitamos comprobar si la consulta tiene algún registro.

Una manera de enfocar este escenario es utilizando el *operador morsa* teniendo en cuenta que `fetchone()` devuelve `None` si la consulta es vacía. Veamos su implementación:

```
>>> con = sqlite3.connect(db_path)
>>> cur = con.cursor()

>>> # Consulta vacía
>>> res = cur.execute('SELECT * FROM pyversions WHERE branch=4.0')

>>> if row := res.fetchone():
...     print(row)
... else:
...     print('Empty query')
...
Empty query

>>> # Consulta con datos
>>> res = cur.execute('SELECT * FROM pyversions WHERE branch=3.0')

>>> if row := res.fetchone():
...     print(row)
... else:
...     print('Empty query')
...
('3.0', 2008, 12, 'Barry Warsaw')
```

## 8.1.6 Otras funcionalidades

### Tablas en memoria

Existe la posibilidad de trabajar con tablas en memoria sin necesidad de tener un fichero en disco.

Veamos un ejemplo muy sencillo:

```
>>> con = sqlite3.connect(':memory:')

>>> cur = con.cursor()

>>> sql = 'CREATE TABLE temp (id INTEGER PRIMARY KEY, value TEXT)'
>>> cur.execute(sql)
```

(continué en la próxima página)

(proviene de la página anterior)

```
<sqlite3.Cursor at 0x107884ea0>

>>> sql = 'INSERT INTO temp VALUES (1, "X")'
>>> cur.execute(sql)
<sqlite3.Cursor at 0x107884ea0>

>>> for row in cur.execute('SELECT * FROM temp'):
...     print(row)
...
(1, 'X')
```

**Prudencia:** Obviamente si no guardamos estos datos los perderemos al no disponer de persistencia.

### Claves autoincrementales

Es muy habitual encontrar en la definición de una tabla un **campo identificador numérico entero** que actúe como clave primaria y se le asignen valores automáticamente.

Existe una forma sencilla de aplicar este escenario en SQLite:

1. Definimos una columna de tipo `INTEGER PRIMARY KEY`.
2. En cualquier operación de inserción, si no especificamos un valor explícito para dicha columna, se rellenará automáticamente con un entero sin usar, típicamente uno más que el último valor generado.

Veamos un ejemplo de aplicación con una tabla en memoria que almacena **ciudades y sus geolocalizaciones**:

```
>>> con = sqlite3.connect(':memory:')
>>> cur = con.cursor()

>>> cur.execute('''CREATE TABLE cities (
... id INTEGER PRIMARY KEY,
... city TEXT UNIQUE,
... latitude REAL,
... longitude REAL)''')
<sqlite3.Cursor at 0x107139bc0>

>>> cur.execute('''INSERT INTO
... cities (city, latitude, longitude)  # Obviamos "id"
... VALUES ("Tokyo", 35.652832, 139.839478)''')
```

(continué en la próxima página)

(proviene de la página anterior)

```
<sqlite3.Cursor at 0x107139bc0>

>>> result = cur.execute('SELECT * FROM cities')
>>> result.fetchall()
[(1, 'Tokyo', 35.652832, 139.839478)]

>>> cur.execute('''INSERT INTO
... cities (city, latitude, longitude)  # Obviamos "id"
... VALUES ("Barcelona", 41.390205, 2.154007)''')
<sqlite3.Cursor at 0x107139bc0>

>>> result = cur.execute('SELECT * FROM cities')
>>> result.fetchall()
[(1, 'Tokyo', 35.652832, 139.839478), (2, 'Barcelona', 41.390205, 2.154007)]
```

**Importante:** Si la clave primaria de una tabla es una columna de tipo `INTEGER` ésta se convierte en un alias para *rowid*.

## Copias de seguridad

Es posible realizar copias de seguridad de manera programática[3]:

```
>>> def progress(status, remaining, total):
...     print(f'Copied {total-remaining} of {total} pages...')
...

>>> src = sqlite3.connect('python.db')
>>> dst = sqlite3.connect('backup.db')

>>> with dst:
...     src.backup(dst, pages=1, progress=progress)
...
Copied 1 of 3 pages...
Copied 2 of 3 pages...
Copied 3 of 3 pages...

>>> dst.close()
>>> src.close()
```

Podemos comprobar que ambas bases de datos tienen el mismo contenido:

[3] Ejemplo tomado de la documentación oficial de Python.

```
>>> src = sqlite3.connect('python.db')
>>> dst = sqlite3.connect('backup.db')

>>> with src, dst:
...     src_cur = src.cursor()
...     dst_cur = dst.cursor()
...     sql = 'SELECT * FROM pyversions'
...     src_data = src_cur.execute(sql).fetchall()
...     dst_data = dst_cur.execute(sql).fetchall()
...     if src_data == dst_data:
...         print('Contents from both DBs are the same!')
...
Contents from both DBs are the same!
```

Un par de incisos respecto a este mecanismo:

1. Funciona incluso si la base de datos está siendo accedida por otros clientes o concurrentemente por la misma conexión.
2. Funciona incluso entre bases de datos `:memory:` y bases de datos en disco.

**Ver también:**

Hacer directamente una copia del fichero `file.db` (desde el propio sistema operativo) también es una opción rápida para disponer de copias de seguridad.

### Información de filas

Cuando ejecutamos una sentencia de modificación sobre la base de datos podemos obtener el **número de filas modificadas**.

Este dato lo sacamos del atributo `rowcount` del cursor. Veamos un ejemplo:

```
>>> con = sqlite3.connect('python.db')
>>> cur = con.cursor()

>>> cur.execute('SELECT * FROM pyversions').fetchall()
[('2.6', 2008, 10, 'Barry Warsaw'),
 ('2.7', 2010, 7, 'Benjamin Peterson'),
 ('3.0', 2008, 12, 'Barry Warsaw'),
 ('3.1', 2009, 6, 'Benjamin Peterson'),
 ('3.2', 2011, 2, 'Georg Brandl'),
 ('3.3', 2012, 9, 'Georg Brandl'),
 ('3.4', 2014, 3, 'Larry Hastings'),
 ('3.5', 2015, 9, 'Larry Hastings'),
 ('3.6', 2016, 12, 'Ned Deily'),
 ('3.7', 2018, 6, 'Ned Deily'),
```

(continué en la próxima página)

(proviene de la página anterior)

```
('3.8', 2019, 10, 'Łukasz Langa'),
('3.9', 2020, 10, 'Łukasz Langa'),
('3.10', 2021, 10, 'Pablo Galindo Salgado'),
('3.11', 2022, 10, 'Pablo Galindo Salgado'),
('3.12', 2023, 10, 'Thomas Wouters'),
('3.13', 2024, 10, 'Thomas Wouters')]

>>> cur.execute('UPDATE pyversions SET released_at_year=2000')
<sqlite3.Cursor at 0x105593dc0>

>>> cur.rowcount
16  # filas modificadas
```

Igualmente cuando insertamos un registro en la base de datos podemos obtener cuál es el **identificador de la últila fila insertada**:

```
>>> cur.execute('INSERT INTO pyversions VALUES ("3.14", 2025, 10, "Guido Van Rossum")
↪')
<sqlite3.Cursor at 0x105593dc0>

>>> cur.lastrowid
17
```

### Ejecución de scripts

¿Qué pasaría si intentamos ejecutar **varias sentencias SQL a la vez** con las herramientas que hemos visto hasta ahora?

Supongamos una tabla de ejemplo que mantiene estadísticas de los mejores jugadores históricos de la NBA. Queremos crear la tabla e insertar 3 registros en una misma ejecución:

```
>>> con = sqlite3.connect(':memory:')

>>> cur = con.cursor()

>>> sql = '''
... CREATE TABLE nba (
...     player TEXT PRIMARY KEY,
...     points INTEGER
... );
... INSERT INTO nba VALUES ('LeBron James', 8023);
... INSERT INTO nba VALUES ('Michael Jordan', 5987);
... INSERT INTO nba VALUES ('Kareem Abdul-Jabbar', 5762);
... '''
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> cur.execute(sql)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ProgrammingError: You can only execute one statement at a time.
```

**Obtenemos un error** indicando que sólo se puede ejecutar una sentencia cada vez.

Para resolver este problema disponemos de la función executescript() que permite ejecutar varias sentencias SQL de una sola vez:

```
>>> cur.executescript(sql)
<sqlite3.Cursor at 0x1028ce840>
```

Aparentemente ahora sí que ha ido todo bien. Podemos comprobar que la tabla está creada y los registros insertados:

```
>>> sql = 'SELECT * FROM nba'
>>> res = cur.execute(sql)

>>> res.fetchall()
[('LeBron James', 8023),
 ('Michael Jordan', 5987),
 ('Kareem Abdul-Jabbar', 5762)]
```

---

## EJERCICIOS DE REPASO

1. Escriba una clase `ToDo` y una clase `Task` que permita implementar una aplicación de gestión de tareas.
2. Escriba una clase `Twitter` junto a dos clases `User` y `Tweet` que permita implementar una aplicación de tipo «Twitter».

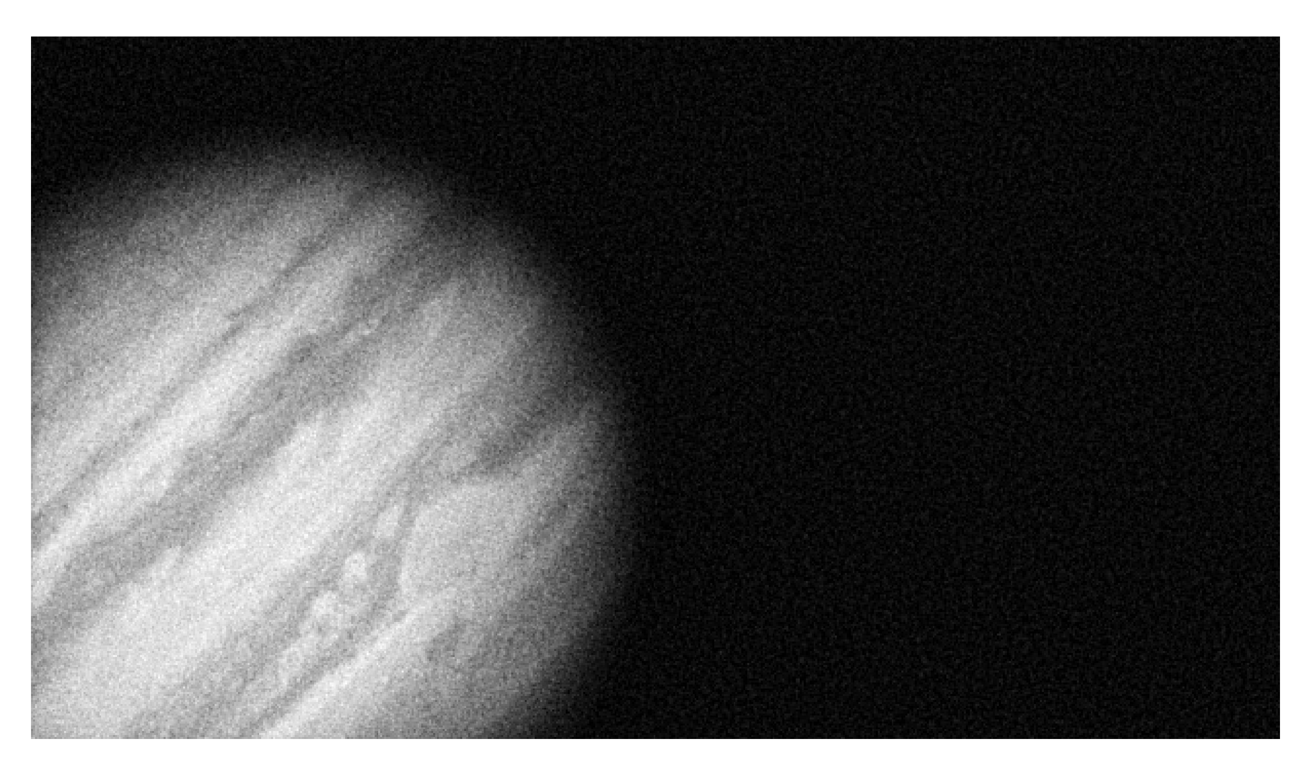
# CAPÍTULO 9

# Ciencia de datos

La **Ciencia de Datos** representa uno de los ámbitos de aplicación más importantes dentro del mundo Python. De hecho, en la encuesta a desarrolladores/as de JetBrains del año 2021 se puede observar que **análisis de datos** y **aprendizaje automático** están en primera y cuarta opción en la pregunta *¿para qué utiliza Python?*.

Muchos de los paquetes que veremos en esta sección también vienen incluidos por defecto en *Anaconda* una plataforma para desarrollo de ciencia de datos que dispone de instaladores para todos los sistemas operativos.

## 9.1 jupyter



El módulo jupyter proporciona un entorno de desarrollo integrado para ciencia de datos, que no es exclusivo de Python, sino que además admite otros lenguajes en su «backend».[1]

```
$ pip install jupyter
```

Para lanzar el servidor de «notebooks»[2]:

```
$ jupyter notebook
```

**Nota:** Este comando nos debería abrir una ventana en el navegador web por defecto del sistema, apuntando a la dirección http://localhost:8888

[1] Foto original de portada por NASA en Unsplash.
[2] Un «notebook» es el concepto de cuaderno (documento) científico que se maneja en Jupyter

### 9.1.1 Notebooks

Un «notebook» es un documento que está compuesto por **celdas** en las que podemos incluir:

- Texto en formato markdown (incluyendo *fórmulas*).
- Elementos multimedia.
- Código Python *ejecutable*.

Figura 1: Ejecución de celdas en Jupyter Notebook

En código «markdown», la salida de la celda es la renderización del texto. En código Python, la salida de la celda es el resultado de la última sentencia incluida en la celda.

**Nota:** Los «notebooks» o cuadernos son básicamente archivos de texto en formato *json* con extensión `.ipynb` (que proviene de «IPython Notebook»).

### 9.1.2 Interfaz

Jupyter se presenta como una aplicación web en cuya interfaz podemos encontrar distintos elementos que nos permitirán desarrollar nuestras tareas de programación de una forma más cómoda.

#### Explorador de archivos

Lo primero que veremos al arrancar el servidor de «notebooks» será el **explorador de archivos** con un diseño muy similar al de cualquier sistema operativo.

**Nota:** Los «notebooks» que se están ejecutando suelen tener un color verde en el icono, mientras que los que están parados aparecen en gris.

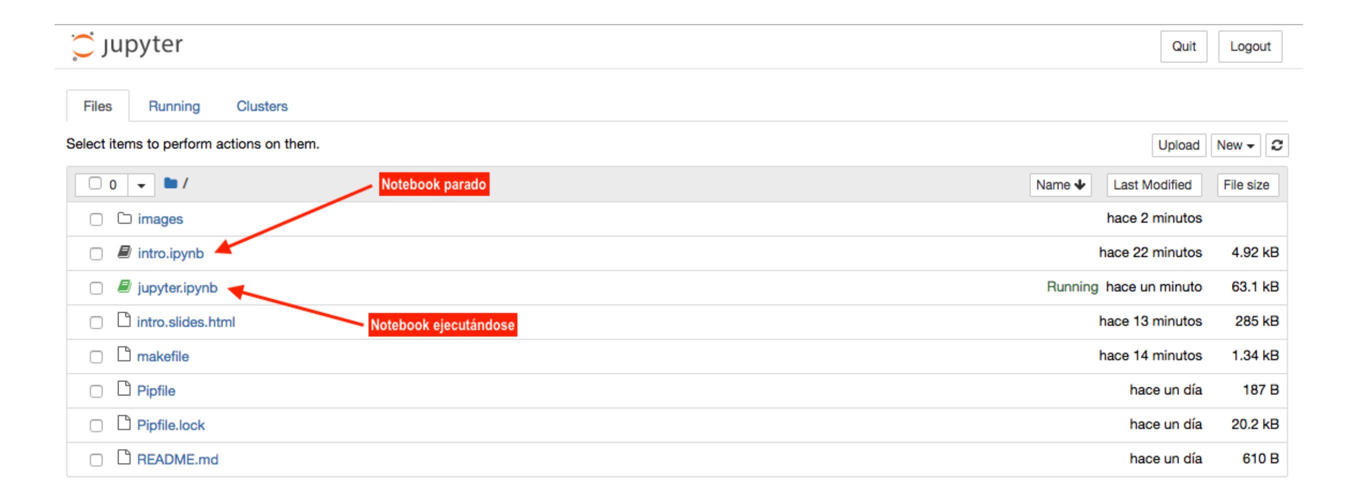



Figura 2: Explorador de archivos de Jupyter Notebook

## Barra de menú

### Menú Fichero

Del estilo de los menús tradicionales de aplicaciones, aquí podemos encontrar las principales funciones sobre ficheros.

**Checkpoints:** Permiten guardar el estado del «notebook» en un momento determinado para luego poder revertirlo a ese momento del tiempo.

**Exportar notebooks:** Es posible exportar «notebooks» a una gran variedad de formatos:

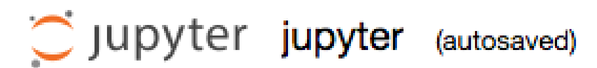
- Python (`.py`)
- HTML (`.html`)
- Reveal.js «slides» (`.html`)
- Markdown (`.md`)
- reST (`.rst`)
- PDF vía LaTeX (`.pdf`)
- asciidoc (`.asciidoc`)
- custom (`.txt`)
- LaTeX (`.tex`)

---

**Ejercicio**

Cree un «notebook» de prueba y descárgelo en formato **HTML** y **Markdown**.

---



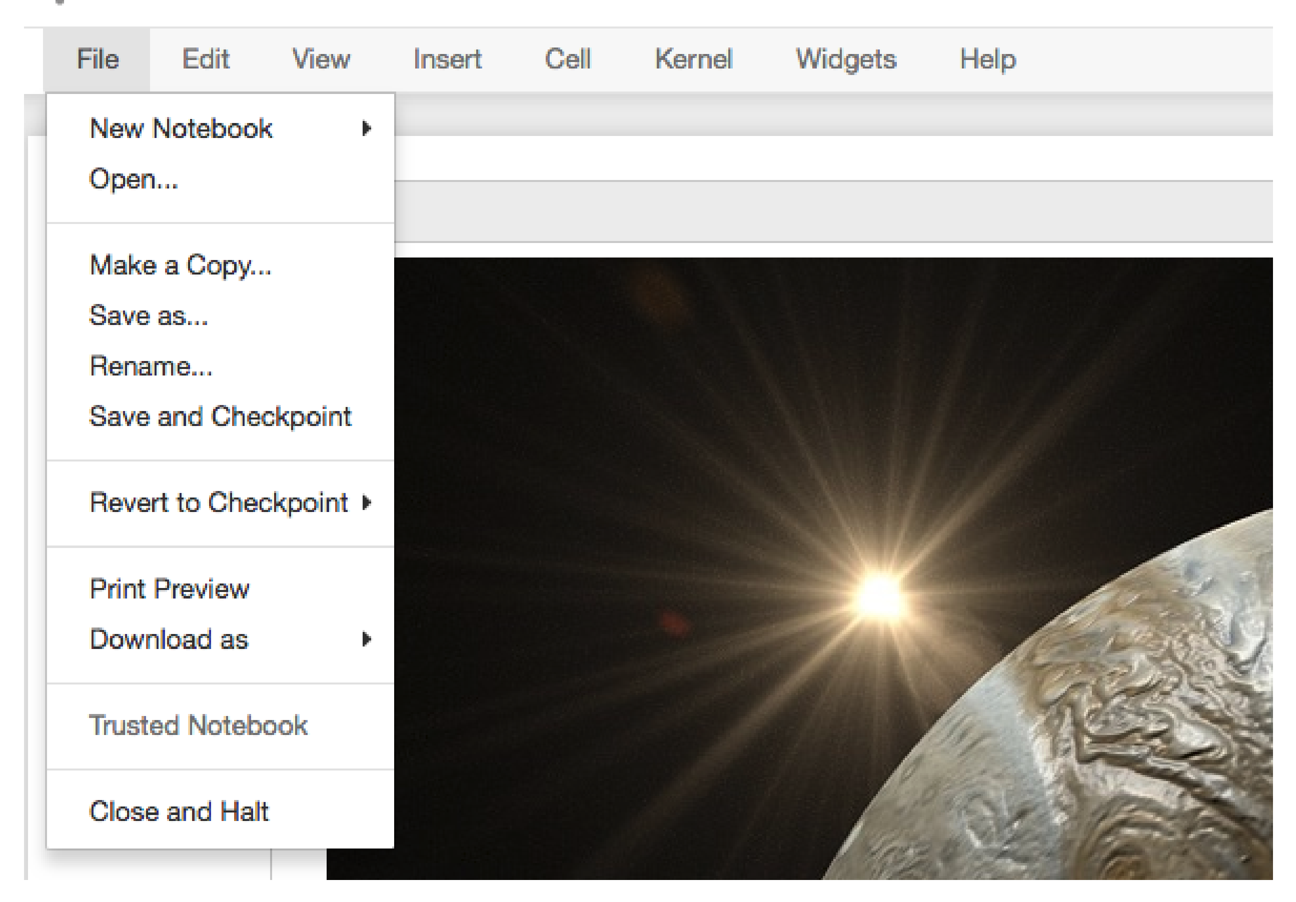




Figura 3: Menú Fichero de Jupyter Notebook

### Menú Edición

Este menú contiene las acciones que podemos realizar sobre una o varias celdas.

Las funciones las podríamos agrupar en **gestión de celdas** (cortar, pegar, borrar, dividir, unir, mover, etc.) e **inserción de imágenes** seleccionando desde un cuadro de diálogo.

### Menú Vista

Permite modificar el aspecto visual de determinados elementos de la aplicación.

**Números de línea:** Puede resultar interesante mostrar los números de línea en celdas que contengan código.

**Modo presentación (Cell Toolbar ▶️ Slideshow) :** Jupyter Notebook ofrece la posibilidad de crear una presentación sobre el documento en el que estamos trabajando. Cada celda se puede configurar con alguno de los siguientes tipos:

- Slide.
- Subslide.
- Fragment.
- Skip.
- Notes.

**Etiquetas (Cell Toolbar ▶️ Tags):** Es interesante – entre otras – el uso de la etiqueta `raises-exception` ya que nos permite ejecutar todas las celdas de un «notebook» sin que el sistema se detenga por errores en la celda etiquetada, ya que estamos informando que lanzará una *excepción*.

### Menú Insertar

Insertar celda antes o después de la actual.

### Menú Celda

Principalmente enfocado a la ejecución de las celdas que componen el «notebook».

**Ejecución de celdas:** La ejecución de celdas se puede hacer de forma individual o grupal así como indicando el punto de partida (celda actual).

**Tipo de celdas:**

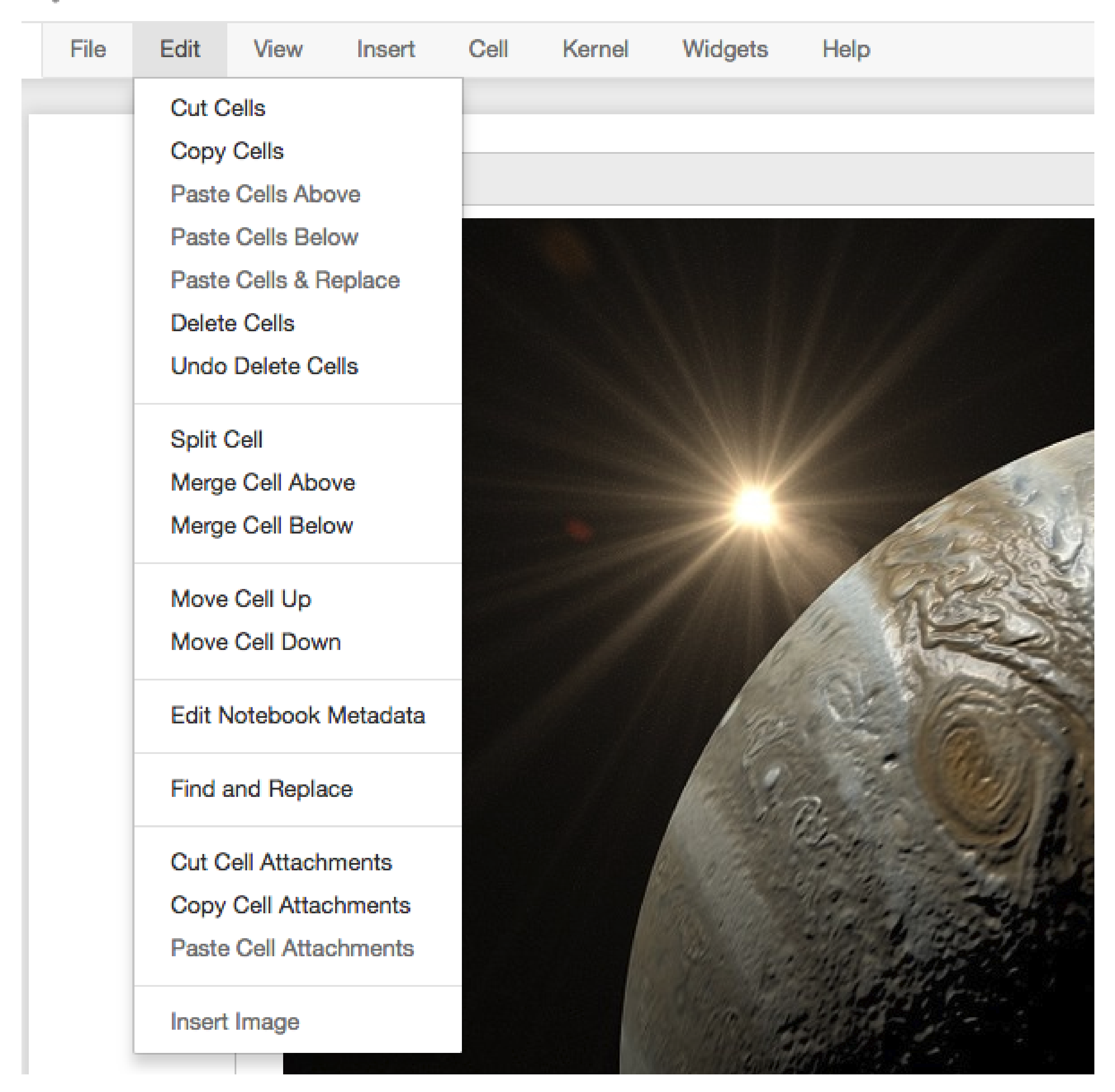



Figura 4: Menú Edición de Jupyter Notebook



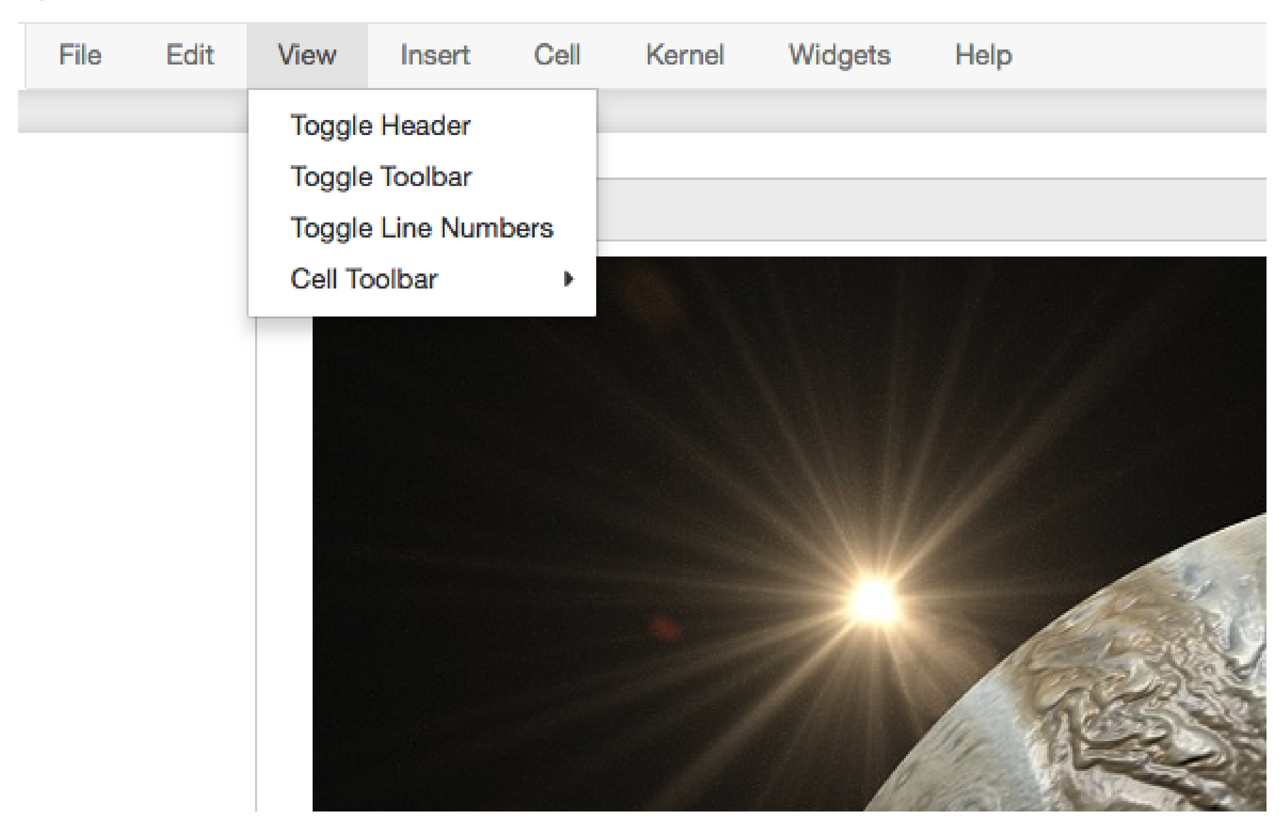




Figura 5: Menú Vista de Jupyter Notebook

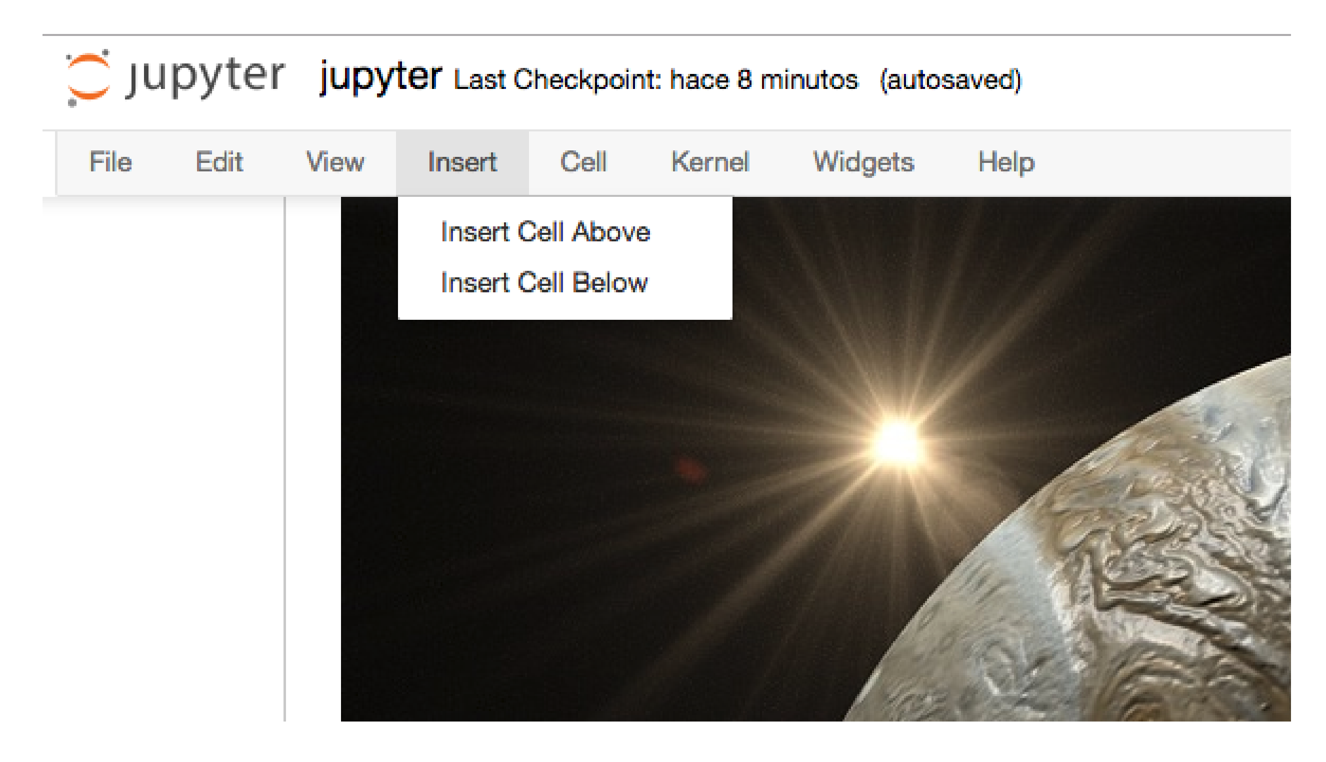



Figura 6: Menú Insertar de Jupyter Notebook

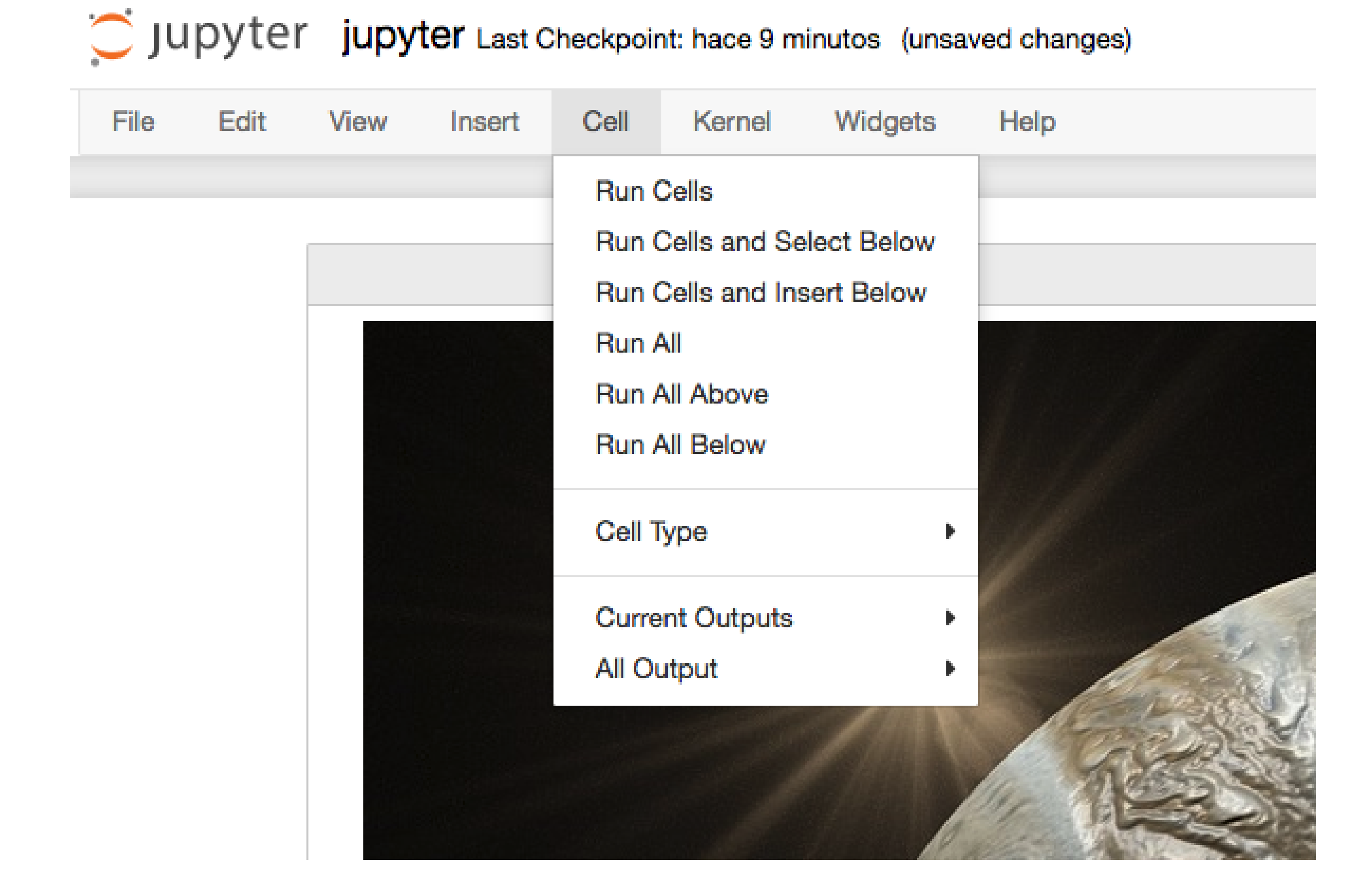



Figura 7: Menú Celda de Jupyter Notebook

- **Code**: para incluir código (se podrá ejecutar el lenguaje de programación según el «kernel» instalado).
- **Markdown**: para escribir texto utilizando sintaxis markdown.
- **Raw**: estas celdas no serán formateadas.

**Salida de celdas:** La ejecución de las celdas de código tiene (suele tener) una salida. Esta salida se puede ocultar (si interesa). Incluso tenemos control sobre activar o desactivar el «scroll» en caso de que la salida sea muy larga.

### Menú Kernel

Permite gestionar el servicio que se encarga de lanzar los «notebooks».



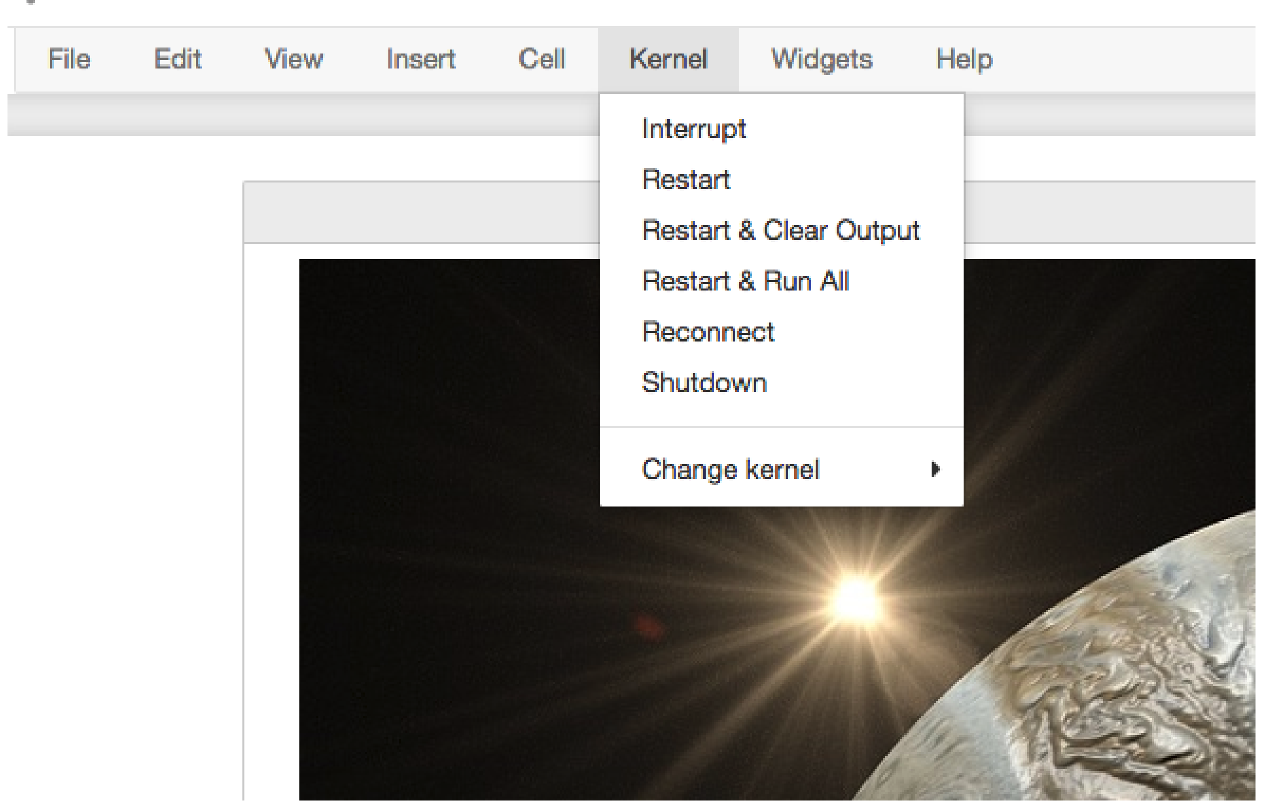




Figura 8: Menú Kernel de Jupyter Notebook

El **kernel** es la capa de software que se encarga de ejecutar las celdas de nuestro «notebook» que contienen código. Podemos tener instalados distintos «kernels» para un mismo Jupyter Notebook. El kernel se puede interrumpir o reiniciar.

Hay veces, que debido a un error de programación o a procesos muy largos, podemos encontrarnos con el «kernel» bloqueado durante un largo período de tiempo. En estas

ocasiones es útil reiniciarlo para salvar esa situación.

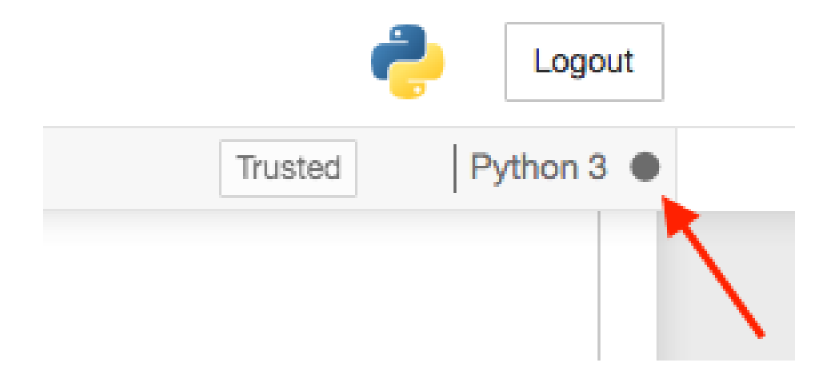



Figura 9: Kernel ocupado

#### Menú Ayuda

Como cualquier aplicación, existe un menú de ayuda en el que se pueden encontrar enlaces a referencias y manuales.

Uno de los elementos más interesantes de la ayuda es el uso de los «shortcuts»[3]. Aunque hay muchos, dejamos aquí algunos de los más útiles:

| Shortcut | Acción |
|---|---|
| `SHIFT + ENTER` | Ejecutar la celda actual |
| `ALT + ENTER` | Ejecutar la celda actual y «abrir» una celda debajo |
| a | Abrir una celda encima de la actual («above») |
| b | Abrir una celda debajo de la actual («below») |
| m | Convertir la celda actual a Markdown |
| y | Convertir la celda actual a código |
| dd | Borrar la celda actual |

### 9.1.3 MathJax

MathJax es una biblioteca javascript que permite visualizar fórmulas matemáticas en navegadores web, utilizando (entre otros) el lenguajes de marcado LaTeX. Para escribir fórmulas matemáticas la celda debe ser de tipo Markdown y tendremos que usar delimitadores especiales.

**Fórmulas «en línea»:** Se debe usar el delimitador dólar antes y después de la expresión `$ ... $`

Por ejemplo: `$ \sum_{x=1}^n sin(x) + cos(x) $` produce :$\sum_{x=1}^{n} sin(x) + cos(x)$

[3] Un «shortcut» es un «atajo de teclado» (combinación de teclas) para lanzar una determinada acción.

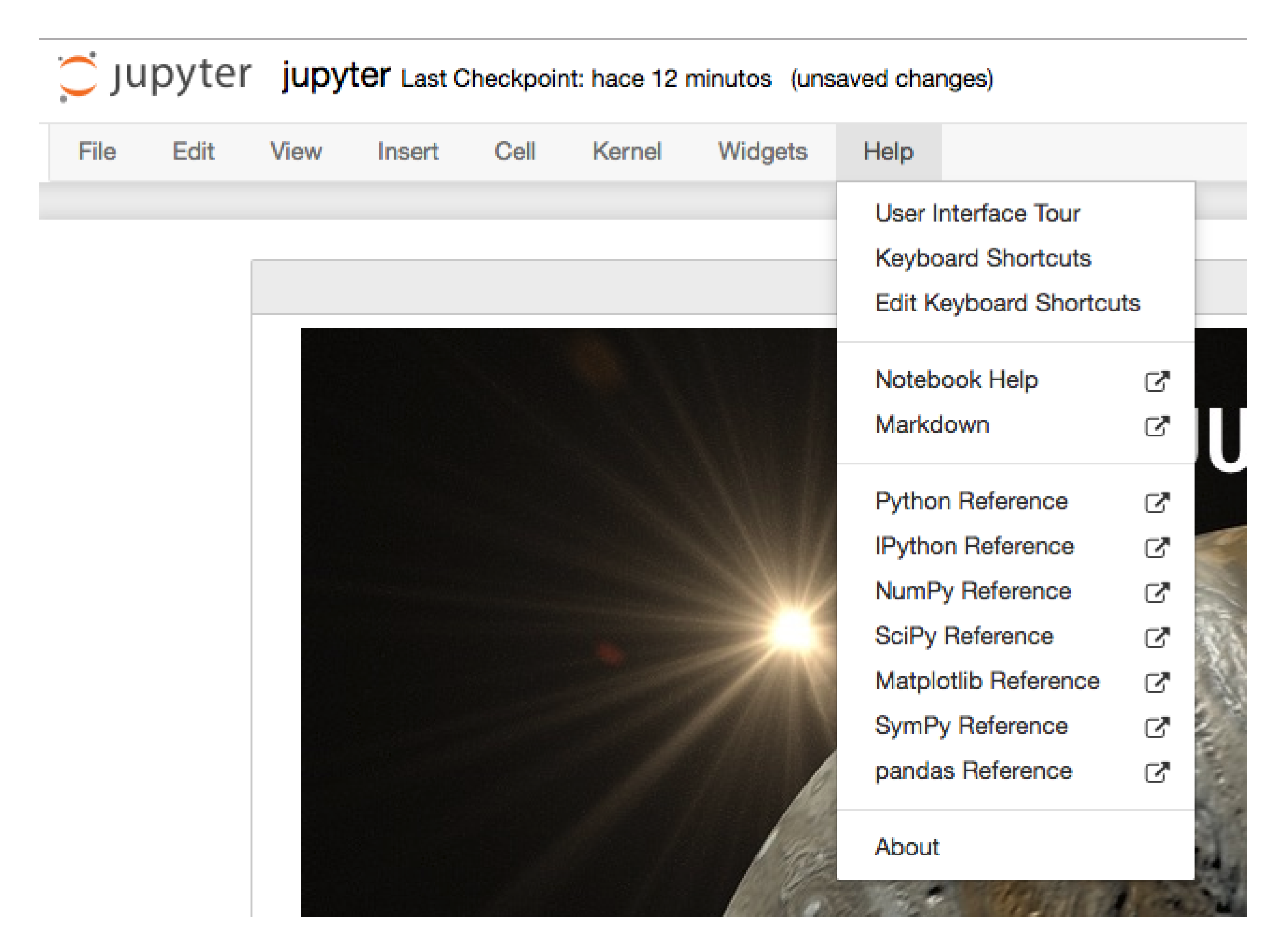



Figura 10: Menú Ayuda de Jupyter Notebook

**Fórmulas «de bloque»:** Se debe usar el delimitador doble dólar antes y después de la expresión `$$ ... $$`

Por ejemplo: `$$ \sum_{x=1}^n sin(x) + cos(x) $$` produce:

$$\sum_{x=1}^{n} sin(x) + cos(x)$$

## Ejemplos de fórmulas

A continuación veremos distintas fórmulas inspiradas en Motivating Examples de la documentación oficial de Jupyter Notebook. Nótese que aunque no se estén indicando los delimitadores `$$` sí habría que ponerlos para conseguir el efecto deseado.

Ecuaciones en varias líneas:

```
\dot{x} = \sigma(y-x) \\
\dot{y} = \rho x - y - xz \\
\dot{z} = -\beta z + xy
```

$$\dot{x} = \sigma(y-x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

Ecuaciones en varias líneas (con alineación):

```
\begin{align}
    \dot{x} &= \sigma(y-x) \\
    \dot{y} &= \rho x - y - xz \\
    \dot{z} &= -\beta z + xy
\end{align}
```

$$\begin{aligned}\dot{x} &= \sigma(y-x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy\end{aligned}$$

Usando paréntesis:

```
\left( \sum_{k=1}^n a_k b_k \right)^2 \leq
\left( \sum_{k=1}^n a_k^2 \right) \left( \sum_{k=1}^n b_k^2 \right)
```

$$\left( \sum_{k=1}^n a_k b_k \right)^2 \leq \left( \sum_{k=1}^n a_k^2 \right) \left( \sum_{k=1}^n b_k^2 \right)$$

Trabajando con matrices:

```
\mathbf{V}_1 \times \mathbf{V}_2 =
\begin{vmatrix}
    \mathbf{i} & \mathbf{j} & \mathbf{k} \\
    \frac{\partial X}{\partial u} &  \frac{\partial Y}{\partial u} & 0 \\
    \frac{\partial X}{\partial v} &  \frac{\partial Y}{\partial v} & 0
\end{vmatrix}
```

$$\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial X}{\partial u} & \frac{\partial Y}{\partial u} & 0 \\ \frac{\partial X}{\partial v} & \frac{\partial Y}{\partial v} & 0 \end{vmatrix}$$

Algo de probabilidad:

```
P(E) = {n \choose k} p^k (1-p)^{ n-k}
```

$$P(E) = {n \choose k} p^k (1-p)^{n-k}$$

Algunos ejemplos con fracciones:

```
\frac{1}{\Bigl(\sqrt{\phi \sqrt{5}}-\phi\Bigr) e^{\frac25 \pi}} =
1+\frac{e^{-2\pi}} {1+\frac{e^{-4\pi}} {1+\frac{e^{-6\pi}}
{1+\frac{e^{-8\pi}} {1+\ldots} } } }
```

$$\frac{1}{\Bigl(\sqrt{\phi \sqrt{5}}-\phi\Bigr) e^{\frac25 \pi}} = 1+\frac{e^{-2\pi}} {1+\frac{e^{-4\pi}} {1+\frac{e^{-6\pi}} {1+\frac{e^{-8\pi}} {1+\ldots} } } }$$

```
1 +  \frac{q^2}{(1-q)}+\frac{q^6}{(1-q)(1-q^2)}+\cdots =
\prod_{j=0}^{\infty}\frac{1}{(1-q^{5j+2})(1-q^{5j+3})},
\quad\quad \text{for $|q|<1$}.
```

$$1 + \frac{q^2}{(1-q)}+\frac{q^6}{(1-q)(1-q^2)}+\cdots = \prod_{j=0}^{\infty}\frac{1}{(1-q^{5j+2})(1-q^{5j+3})}, \quad\quad \text{for $|q|<1$}.$$

Múltiples puntos de alineación:

```
\begin{eqnarray}
    x' &=&   &x \sin\phi &+& z \cos\phi \\
    z' &=& - &x \cos\phi &+& z \sin\phi
\end{eqnarray}
```

$$\begin{array}{rcrlcl} x' &=& & x \sin\phi &+& z \cos\phi \\ z' &=& - & x \cos\phi &+& z \sin\phi \end{array}$$

## Ejercicio

Escriba en MathJax las siguientes ecuaciones:

**Ecuación 1**

$$\int_a^b f'(x)dx = f(b) - f(a)$$

**Ecuación 2**

$$t' = t\frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

**Ecuación 3**

$$\left[M\frac{\partial}{\partial M} + \beta(g)\frac{\partial}{\partial g} + \eta\gamma\right]G^n(x_1, x_2, \ldots, x_n; M, g) = 0$$

**Ecuación 4**

$$R_{00} \approx -\frac{1}{2}\sum_i \frac{\partial^2 h_{00}}{\partial (x^i)^2} = \frac{4\pi G}{c^2}(\rho c^2) \Rightarrow \bigtriangledown^2 \phi_g = 4\pi G\rho$$

---

**Truco:** Puede encontrar símbolos matemáticos para Latex en este enlace así como dibujar directamente un símbolo y obtener su referencia a través de la herramienta Detexify.

---

### 9.1.4 Comandos especiales

Jupyter Notebook ofrece una gama de comandos especiales que cubren gran variedad de funcionalidades.

#### Comandos de shell

Podemos ejecutar comandos de «shell» usando el prefijo exclamación `!`

```
>>> !date
martes, 15 de junio de 2021, 09:13:25 WEST
```

```
>>> !whoami
sdelquin
```

---

**Ejercicio**

Ejecute los siguientes comandos del sistema y obtenga la salida en una celda del Notebook:

| Windows | Linux & macOS |
|---|---|
| `time` | `date` |
| `dir` | `ls` |
| `mem` | `free` |

## Obteniendo ayuda

Una de las formas más sencillas de obtener información de librerías, funciones o módulos es utilizar el sufijo interrogación `?`

```
>>> import random

>>> random.randint?
Signature: random.randint(a, b)
Docstring:
Return random integer in range [a, b], including both end points.

File:      ~/.pyenv/versions/3.9.1/lib/python3.9/random.py
Type:      method
```

### Ejercicio

Obtenga la documentación de las siguientes funciones:

- `os.path.dirname`
- `re.match`
- `datetime.timedelta`

## Comandos mágicos

Jupyter Notebook, o mejor expresado IPython, admite un conjunto de comandos mágicos que permiten realizar distintas tareas, en muchos casos, no necesariamente relacionadas con Python:

```
>>> %lsmagic
Available line magics:
%aimport  %alias  %alias_magic  %autoawait  %autocall  %autoindent  %automagic
↪%autoreload  %bookmark  %cat  %cd  %clear  %colors  %conda  %config  %cp  %cpaste
↪%debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui  %hist  %history
↪%killbgscripts  %ldir  %less  %lf  %lk  %ll  %load  %load_ext  %loadpy  %logoff
↪%logon  %logstart  %logstate  %logstop  %ls  %lsmagic  %lx  %macro  %magic  %man
↪%matplotlib  %mkdir  %more  %mv  %notebook  %page  %paste  %pastebin  %pdb  %pdef
↪%pdoc  %pfile  %pinfo  %pinfo2  %pip  %popd  %pprint  %precision  %prun  %psearch
↪%psource  %pushd  %pwd  %pycat  %pylab  %quickref  %recall  %rehashx  %reload_ext
↪%rep  %rerun  %reset  %reset_selective  %rm  %rmdir  %run  %save  %sc  %set_env
↪%store  %sx  %system  %tb  %time  %timeit  %unalias  %unload_ext  %who  %who_ls
```

(continué en la próxima página)

(proviene de la página anterior)

```
Available cell magics:
%%! %%HTML %%SVG %%bash %%capture %%debug %%file %%html %%javascript %%js %
↪%latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %%ruby␣
↪ %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

Si nos fijamos en el último mensaje, al estar habilitado el modo «automagic», no es estrictamente necesario que usemos el prefijo `%` para hacer uso de estos comandos. Por ejemplo, si quisiéramos conocer la *historia de comandos* en el intérprete:

```
>>> hist  # equivalente a %hist
!date
import random
random.randint?
%lsmagic
pwd
hist
```

## Representando gráficas

Otra de las grandes ventajas que ofrece Jupyter Notebook es poder graficar directamente sobre el cuaderno. Para ello utilizamos código Python (en este caso) y una directiva de comando mágico para indicar que se renderice en línea:

```
>>> %matplotlib inline

>>> from matplotlib import pyplot as plt

>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> y = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> plt.plot(x, y)
[<matplotlib.lines.Line2D at 0x106414e50>]
<Figure size 432x288 with 1 Axes>
```

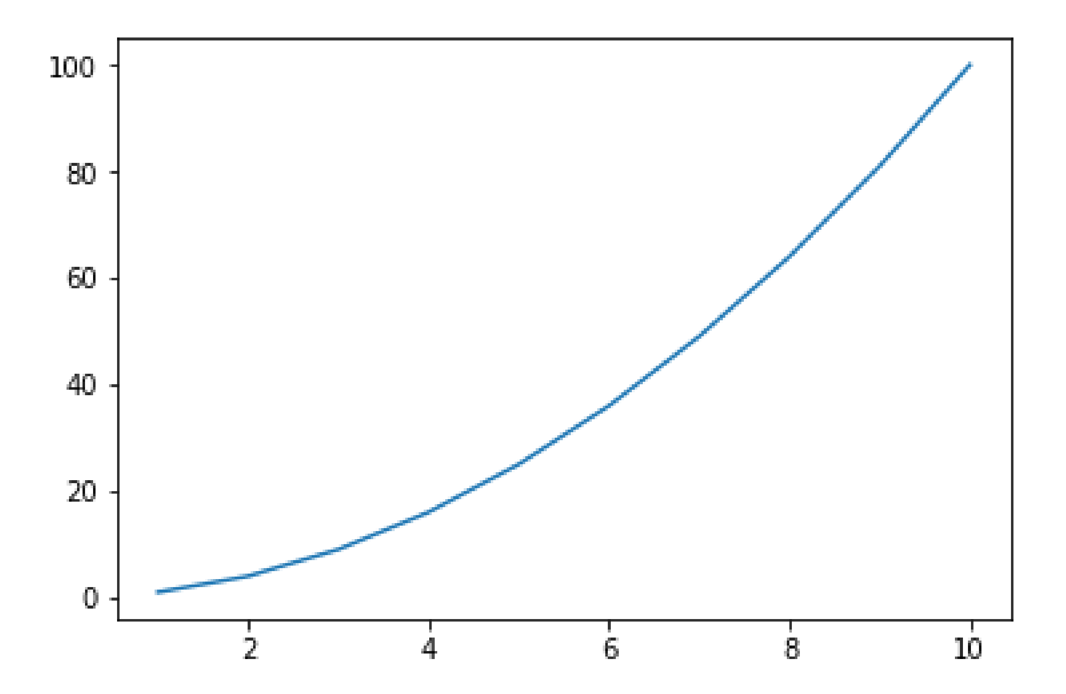



Figura 11: Gráfica sencilla hecha en Jupyter Notebook

## Manejando ficheros

**Cargando un fichero en la celda actual:** Para ello utilizamos el comando `%load "ruta/al/fichero"`

**Ejecutando un fichero en la celda actual:** Para ello utilizamos el comando `%run "ruta/al/fichero"`

**Escribiendo el contenido de la celda actual a fichero:** Para ello utilizamos el comando `%writefile "ruta/al/fichero"` como **primera línea de la celda** y después vendría el código que queremos escribir.

---

**Ejercicio**

- En una celda del «notebook», escriba código Python para crear una lista de 100 números pares.
- Guarde el contenido de esa celda un fichero Python usando `%%writefile`
- Carge este fichero en una celda con `%load`
- Ejecútelo con `%run`

---

### Tiempos de ejecución

Para medir el tiempo de ejecución de una determinada instrucción Python podemos utilizar el comando `%timeit` que calcula un promedio tras correr repetidas veces el código indicado:

```
>>> import numpy

>>> %timeit numpy.random.normal(size=100)
3.03 µs ± 6.77 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

De igual forma, existe un mecanismo para medir el tiempo de ejecución de una celda completa. En este caso se utiliza el comando `%%timeit` (nótese la diferencia del doble porcentaje como prefijo):

```
%%timeit

numpy.random.poisson(size=100)
numpy.random.uniform(size=100)
numpy.random.logistic(size=100)

8.88 µs ± 25.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

---

**Ejercicio**

Mida si hay diferencias significativas en tiempos de ejecución en la creación de distribuciones aleatorias atendiendo a:

- Tipo de distribución (*Poisson*, *Uniform*, *Logistic*).
- Tamaño de la muestra (100, 10000, 1000000).

---

### Incluyendo otros lenguajes

**Celdas con HTML:** Si necesitamos insertar código HTML en una celda, podemos usar el comando `%%html` al comienzo de la misma:

```
%%html

<iframe src="https://www.google.com/maps/embed?pb=!1m18!1m12!1m3!1d3592984.
↪8538165656!2d-18.096789575396794!3d28.426067294993228!2m3!1f0!2f0!3f0!3m2!
↪1i1024!2i768!4f13.1!3m3!1m2!1s0xc41aa86ef755363%3A0x10340f3be4bc8c0!
↪2sCanarias!5e0!3m2!1ses!2ses!4v1623755509663!5m2!1ses!2ses" width="400"␣
↪height="300" style="border:0;" allowfullscreen="" loading="lazy"></iframe>
```




**Celdas con «shell script»:** Hay ocasiones en las que un código en shell script suele ser útil. Para incluirlo recurrimos al comando `%%bash` al principio de la celda:

```
%%bash

!tree -d -L 2
.
├── __pycache__
├── _build
│   └── html
├── _static
│   ├── css
│   ├── img
│   └── js
├── core
│   ├── controlflow
│   ├── datastructures
│   ├── datatypes
│   ├── devenv
│   ├── introduction
│   └── modularity
├── miniprojects
```

(continué en la próxima página)

(proviene de la página anterior)

```
│   ├── spotify
├── pypi
│   ├── datascience
├── stdlib
    ├── text_processing

20 directories
```

**Celdas con perl:** No hay que subestimar el poder del lenguaje de programación perl. Si fuera necesario, lo podemos incluir en una celda del «notebook» con `%%perl` al comienzo de la misma:

```
%%perl

my $email = 'sdelquin@gmail.com';

if ($email =~ /^([^@]+)\@(.+)$/) {
    print "Username is: $1\n";
    print "Hostname is: $2\n";
}

...

Username is: sdelquin
Hostname is: gmail.com
```

### 9.1.5 Extensiones

El ecosistema de Jupyter Notebook es muy amplio y ofrece una gran variedad de extensiones que se pueden incluir en la instalación que tengamos: Unofficial Jupyter Notebook Extensions.

Su instalación es tan sencilla como:

```
$ pip install jupyter_contrib_nbextensions
```

### 9.1.6 Otros entornos

El ecosistema de entornos para trabajos en ciencia de datos ha ido ampliándose durante estos últimos años con la explosión del «BigData» y la inteligencia artificial. En este apartado veremos otras plataformas que también nos permiten usar Python enfocado al análisis de datos.

#### JupyterLab

JupyterLab es una evolución de Jupyter Notebook. Entre sus mejoras podemos destacar:

- Explorador de ficheros integrado en la barra lateral.
- Posibilidad de abrir múltiples `.ipynb` al mismo tiempo usando pestañas.
- Posibilidad de abrir múltiples terminales.
- Editor integrado para cualquier fichero de texto.
- Vista previa en tiempo real de documentos *markdown* o *csv*.

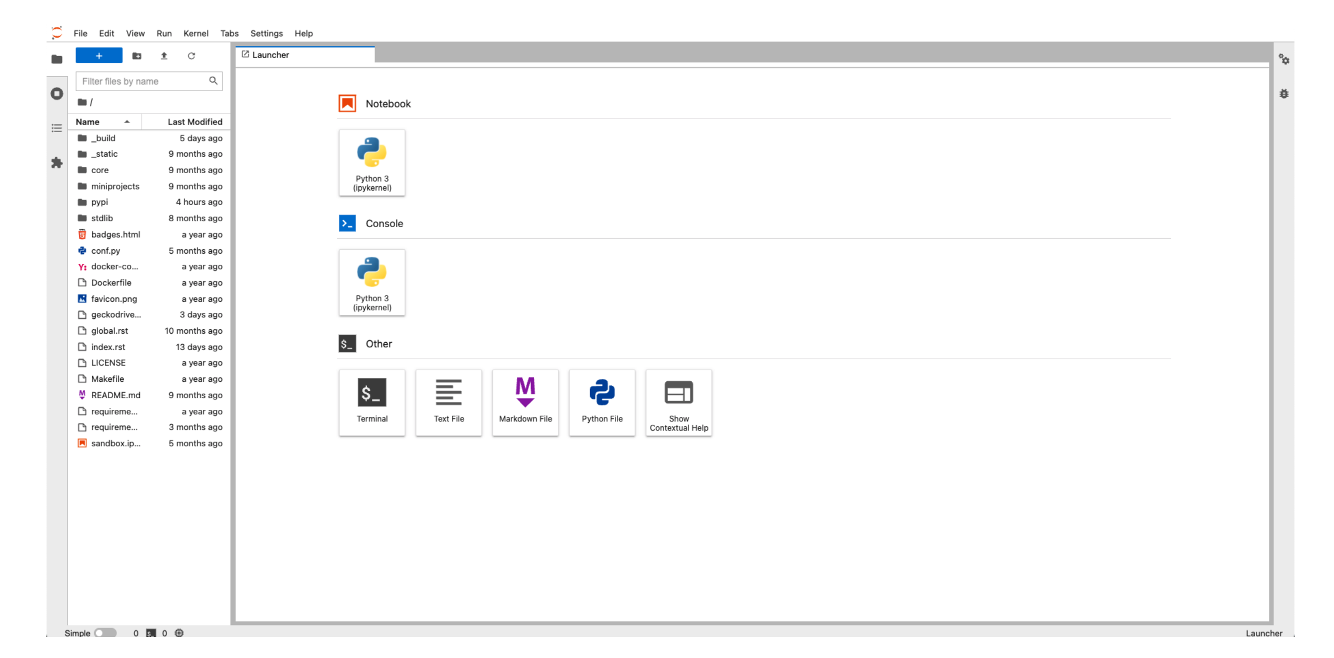



Figura 12: Pantalla inicial de JupyterLab

Su instalación se lleva a cabo como cualquier otro paquete Python:

```
$ pip install jupyterlab
```

Para ejecutar la aplicación:

```
$ jupyter-lab
```

## Google Colab

Google Colab es un entorno de computación científica creado por Google y disponible en su nube. Como era previsible, para su uso es necesario disponer de una cuenta en Google.

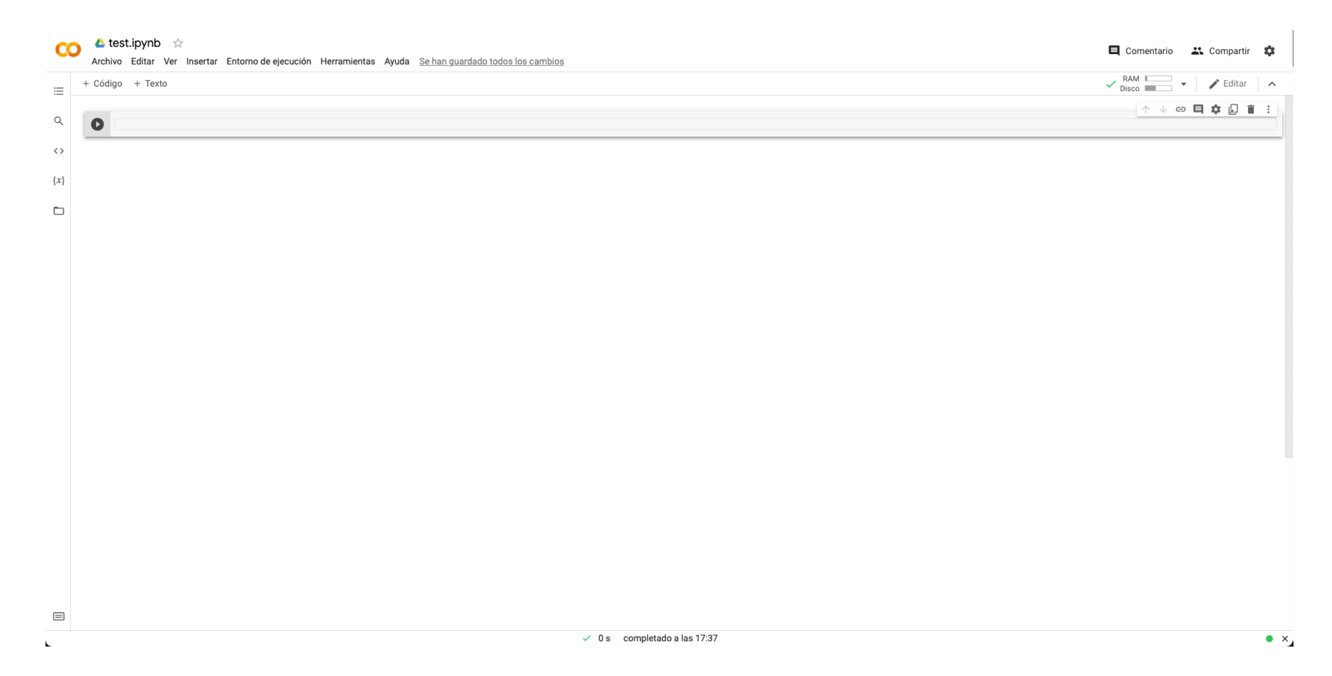



Figura 13: Pantalla inicial de Google Colab

**Características**:

- Tiene un comportamiento totalmente análogo a Jupyter en cuanto a comportamiento y funcionalidades.
- Completamente en la nube. No necesita instalación ni configuración.
- Por defecto trae multitud de paquetes instalados, principalmente en el ámbito científico: 386 paquetes (febrero de 2022).
- Versión de Python: 3.7.12 (febrero de 2022).
- Espacio en disco sujeto a las características de Google Compute Engine: 107.72GB (febrero de 2022)
- Memoria RAM sujeta a las características de Google Compute Engine: 12.69GB (febrero de 2022)
- Acceso limitado al sistema operativo.

- En cuentas gratuitas, los tiempos de cómputo son, por lo general, mayores que en una máquina local.[4]
- Previsualización *markdown* en tiempo real sobre cada celda.
- Posibilidad de subir ficheros de datos propios en carpetas accesibles por el cuaderno.
- Posibilidad de ejecutar Jupyter «notebooks» propios.
- Posibilidad (limitada) de acelerar cálculos usando GPU[6] o TPU[7].
- Posibilidad de descargar el cuaderno como Jupyter «notebook» o archivo de Python.
- Índice de contenidos integrado en barra lateral.
- Inspector de variables integrado en barra lateral.

## Kaggle

Kaggle es una plataforma que no sólo ofrece un entorno de trabajo para cuadernos Jupyter sino también una enorme colección de conjuntos de datos de libre acceso. Para su uso es necesario disponer de una cuenta en el servicio.

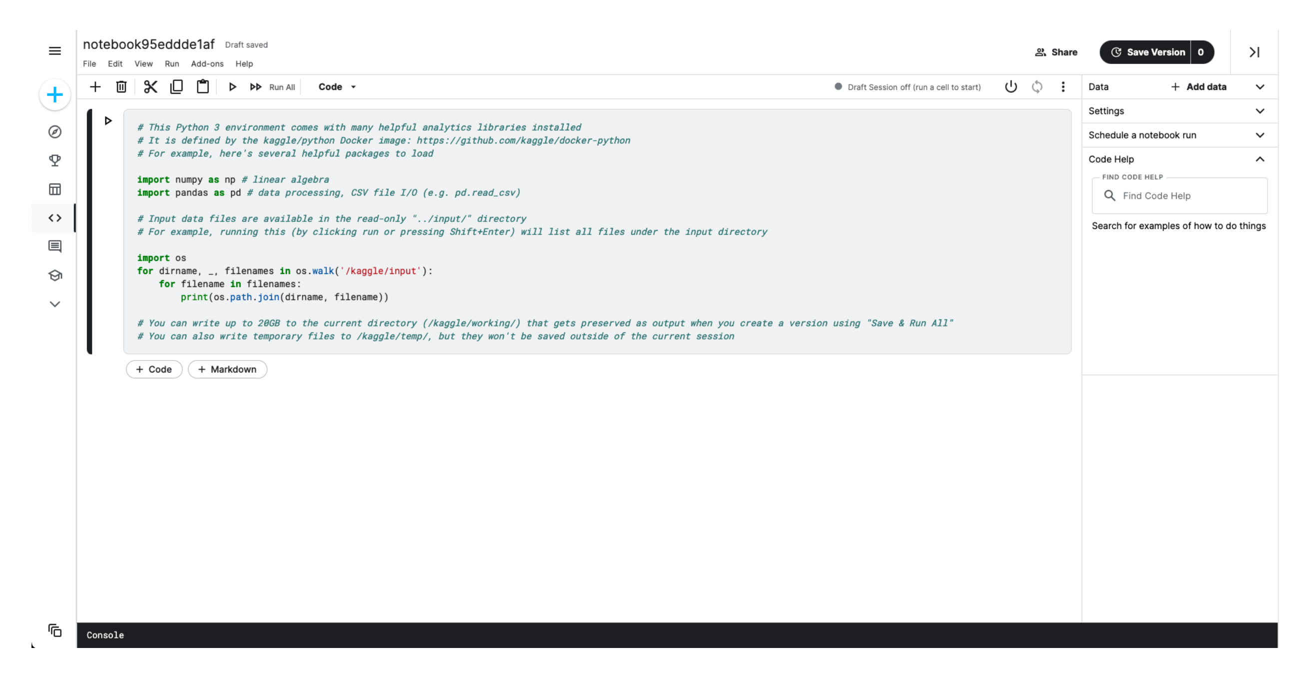



Figura 14: Pantalla inicial de Kaggle

**Características:**

[4] Todo estará en función de las características de la máquina con la que se esté trabajando.
[6] Graphics Processing Unit (Unidad gráfica de procesamiento).
[7] Tensor Processing Unit (Unidad de procesamiento tensorial).

- Tiene un comportamiento totalmente análogo a Jupyter en cuanto a comportamiento y funcionalidades.
- Completamente en la nube. No necesita instalación ni configuración.
- Por defecto trae multitud de paquetes instalados, principalmente en el ámbito científico: 792 paquetes (febrero de 2022).
- Versión de Python: 3.7.12 (febrero de 2022).
- Espacio en disco sujeto a las características de Kaggle: 73.1GB (febrero de 2022)
- Memoria RAM sujeta a las características de Kaggle: 16GB (febrero de 2022)
- Acceso limitado al sistema operativo.
- En cuentas gratuitas, los tiempos de cómputo son, por lo general, mayores que en una máquina local.[4]
- Posibilidad de subir ficheros de datos propios sólo como «datasets» de Kaggle.
- Posibilidad de ejecutar Jupyter «notebooks» propios.
- Posibilidad (limitada) de acelerar cálculos usando GPU[6] o TPU[7].
- Posibilidad de descargar el cuaderno como Jupyter «notebook».

### Comparativa

Haremos una comparativa de tiempos de ejecución lanzando una FFT[5] sobre una matriz de 1 millón de elementos:

```
>>> import numpy as np

>>> bigdata = np.random.randint(1, 100, size=(1_000, 1_000))

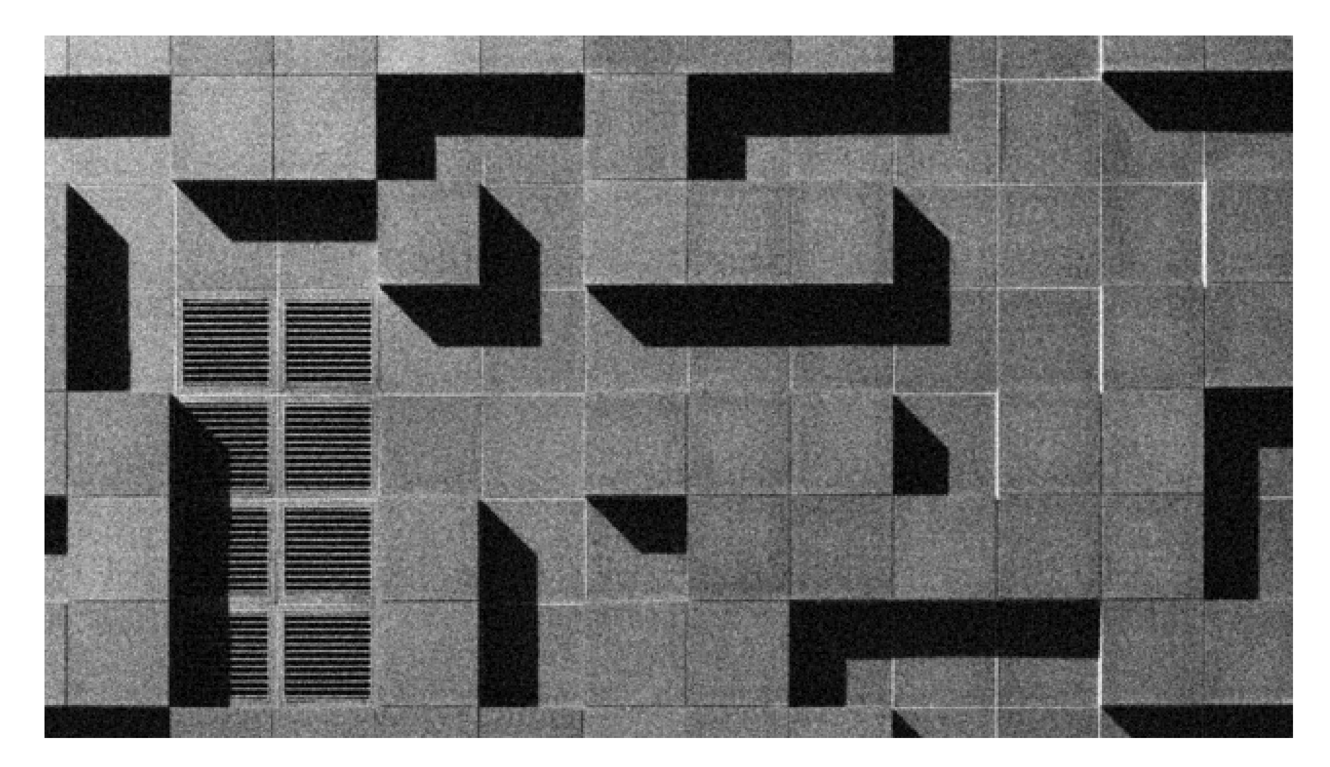
>>> %timeit np.fft.fft(bigdata)
4.89 ms ± 5.78 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

| Jupyter | Colab | Kaggle |
|---|---|---|
| 4.89ms | 13.9ms | 12.8ms |

Obviamente se trata de una ejecución puntual y no podemos sacar conclusiones claras al respecto. Además de ello depende del «hardware» sobre el que estemos trabajando. En cualquier caso el propósito es únicamente tener una ligera idea de los órdenes de magnitud.

[5] Fast Fourier Transform (Transformada rápida de Fourier).

## 9.2 numpy



NumPy es el paquete fundamental para computación científica en Python y manejo de arrays numéricos multi-dimensionales.[1]

```
$ pip install numpy
```

La forma más común de importar esta librería es usar el alias `np`:

```
>>> import numpy as np
```

### 9.2.1 ndarray

En el núcleo de NumPy está el `ndarray`, donde «nd» es por n-dimensional. Un ndarray es un array multidimensional de **elementos del mismo tipo**.

Aquí tenemos una diferencia fundamental con las *listas* en Python que pueden mantener objetos heterogéneos. Y esta característica propicia que el rendimiento de un ndarray sea bastante mejor que el de una lista convencional.

Para crear un array podemos usar:

[1] Foto original de portada por Vlado Paunovic en Unsplash.

```
>>> import numpy as np

>>> x = np.array([1, 2, 3, 4, 5])

>>> x
array([1, 2, 3, 4, 5])

>>> type(x)
numpy.ndarray
```

Si queremos obtener información sobre el array creado, podemos acceder a distintos atributos del mismo:

```
>>> x.ndim  # dimensión
1

>>> x.size  # tamaño total del array
5

>>> x.shape  # forma
(5,)

>>> x.dtype  # tipo de sus elementos
dtype('int64')
```

### Datos heterogéneos

Hemos dicho que los ndarray son estructuras de datos que almacenan un único tipo de datos. A pesar de esto, es posible crear un array con los siguientes valores:

```
>>> x = np.array([4, 'Einstein', 1e-7])
```

Aunque, a priori, puede parecer que estamos mezclando tipos enteros, flotantes y cadenas de texto, lo que realmente se produce (de forma implícita) es una coerción[2] de tipos a **Unicode**:

```
>>> x
array(['4', 'Einstein', '1e-07'], dtype='<U32')

>>> x.dtype
dtype('<U32')
```

[2] Característica de los lenguajes de programación que permite, implícita o explícitamente, convertir un elemento de un tipo de datos en otro, sin tener en cuenta la comprobación de tipos.

### Tipos de datos

NumPy maneja gran cantidad de tipos de datos. A diferencia de los *tipos de datos numéricos en Python* que no establecen un tamaño de bytes de almacenamiento, aquí sí hay una diferencia clara.

Algunos de los tipos de datos numéricos en NumPy se presentan en la siguiente tabla:

Tabla 1: Tipos de datos numéricos en NumPy

| dtype | Descripción | Rango |
|---|---|---|
| np.int32 | Integer | De -2147483648 a 2147483647 |
| np.int64 | Integer | De -9223372036854775808 a 9223372036854775807 |
| np.uint32 | Unsigned integer | De 0 a 4294967295 |
| np.uint64 | Unsigned integer | De 0 a 18446744073709551615 |
| np.float32 | Float | De -3.4028235e+38 a 3.4028235e+38 |
| np.float64 | Float | De -1.7976931348623157e+308 a 1.7976931348623157e+308 |

**Truco:** NumPy entiende por defecto que `int` hace referencia a `np.int64` y que `float` hace referencia a `np.float64`. Son «alias» bastante utilizados.

Si creamos un array de **números enteros**, el tipo de datos por defecto será `int64`:

```
>>> a = np.array(range(10))

>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a.dtype
dtype('int64')
```

Sin embargo podemos especificar el tipo de datos que nos interese:

```
>>> b = np.array(range(10), dtype='int32')  # 'int32' hace referencia a np.int32

>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int32)

>>> b.dtype
dtype('int32')
```

Lo mismo ocurre con **valores flotantes**, donde `float64` es el tipo de datos por defecto.

Es posible convertir el tipo de datos que almacena un array mediante el método `astype`. Por

ejemplo:

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> c = a.astype(float)

>>> c.dtype
dtype('float64')
```

## ndarray vs list

Como ya se ha comentado en la introducción de esta sección, el uso de `ndarray` frente a `list` está justificado por cuestiones de rendimiento. Pero veamos un ejemplo clarificador en el que sumamos 10 millones de valores enteros:

```
>>> array_as_list = list(range(int(10e6)))
>>> array_as_ndarray = np.array(array_as_list)

>>> %timeit sum(array_as_list)
48 ms ± 203 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

>>> %timeit array_as_ndarray.sum()
3.83 ms ± 4.84 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

**Nota:** El cómputo es casi 12 veces más rápido utilizando ndarray frente a listas clásicas.

En cualquier caso, existe la posibilidad de **convertir a lista** cualquier ndarray mediante el método `tolist()`:

```
>>> a = np.array([10, 20, 30])

>>> a
array([10, 20, 30])

>>> b = a.tolist()

>>> b
[10, 20, 30]

>>> type(b)
list
```

### Matrices

Una matriz no es más que un array bidimensional. Como ya se ha comentado, NumPy provee `ndarray` que se comporta como un array multidimensional con lo que podríamos crear una matriz sin mayor problema.

Veamos un ejemplo en el que tratamos de construir la siguiente matriz:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Nos apoyamos en una *lista de listas* para la creación de la matriz:

```
>>> M = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])

>>> M
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

>>> M.ndim  # bidimensional
2

>>> M.size
12

>>> M.shape  # 4 filas x 3 columnas
(4, 3)

>>> M.dtype
dtype('int64')
```

---

**Ejercicio**

Cree los siguientes arrays en NumPy:

$$\text{array1} = \begin{bmatrix} 88 & 23 & 39 & 41 \end{bmatrix}$$

$$\text{array2} = \begin{bmatrix} 76.4 & 21.7 & 38.4 \\ 41.2 & 52.8 & 68.9 \end{bmatrix}$$

$$\text{array3} = \begin{bmatrix} 12 \\ 4 \\ 9 \\ 8 \end{bmatrix}$$

Obtenga igualmente las siguientes características de cada uno de ellos: dimensión, tamaño, forma y tipo de sus elementos.

### Cambiando la forma

Dado un array, podemos cambiar su forma mediante la función `np.reshape()`:

```
>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

>>> np.reshape(a, (3, 4))  # 3 x 4
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Si sólo queremos especificar un número determinado de filas o columnas, podemos dejar la otra dimensión a -1:

```
>>> np.reshape(a, (6, -1))  # 6 filas
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12]])

>>> np.reshape(a, (-1, 3))  # 3 columnas
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

**Advertencia:** En el caso de que no exista posibilidad de cambiar la forma del array por el número de filas y/o columnas especificado, obtendremos un error de tipo `ValueError: cannot reshape array`.

### Almacenando arrays

Es posible que nos interese almacenar (de forma persistente) los arrays que hemos ido creando. Para ello NumPy nos provee, al menos, de dos mecanismos:

**Almacenamiento en formato binario propio:** Mediante el método `save()` podemos guardar la estructura de datos en ficheros .npy. Veamos un ejemplo:

```
>>> M
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

>>> np.save('my_matrix', M)

>>> !ls my_matrix.npy
my_matrix.npy

>>> M_reloaded = np.load('my_matrix.npy')

>>> M_reloaded
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

**Almacenamiento en formato de texto plano:** NumPy proporciona el método `savetxt()` con el que podremos volcar la estructura de datos a un fichero de texto csv. Veamos un ejemplo:

```
>>> M
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

>>> np.savetxt('my_matrix.csv', M, fmt='%d')

>>> !cat my_matrix.csv
1 2 3
4 5 6
7 8 9
10 11 12

>>> M_reloaded = np.loadtxt('my_matrix.csv', dtype=int)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> M_reloaded
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

**Truco:** Por defecto el almacenamiento y la carga de arrays en formato texto usa tipos de **datos flotantes**. Es por ello que hemos usado el parámetro `fmt` en el almacenamiento y el parámetro `dtype` en la carga.

Es posible cargar un array **desempaquetando sus valores** a través del parámetro `unpack`. En el siguiente ejemplo separamos las columnas en tres variables diferentes:

```
>>> M
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

>>> col1, col2, col3 = np.loadtxt('my_matrix.csv', unpack=True, dtype=int)

>>> col1
array([ 1,  4,  7, 10])
>>> col2
array([ 2,  5,  8, 11])
>>> col3
array([ 3,  6,  9, 12])
```

## 9.2.2 Funciones predefinidas para creación de arrays

NumPy ofrece una gran variedad de funciones predefinidas para creación de arrays que nos permiten simplificar el proceso de construcción de este tipo de estructuras de datos.

### Valores fijos

A continuación veremos una serie de funciones para crear arrays con valores fijos.

### Ceros

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Por defecto, ésta y otras funciones del estilo, devuelven **valores flotantes**. Si quisiéramos trabajar con valores enteros basta con usar el parámetro `dtype`:

```
>>> np.zeros((3, 4), dtype=int)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Existe la posibilidad de crear un array de **ceros con las mismas dimensiones** (y forma) que otro array:

```
>>> M = np.array([[1, 2, 3], [4, 5, 6]])

>>> M
array([[1, 2, 3],
       [4, 5, 6]])

>>> np.zeros_like(M)
array([[0, 0, 0],
       [0, 0, 0]])
```

Lo cual sería equivalente a pasar la «forma» del array a la función predefinida de creación de ceros:

```
>>> np.zeros(M.shape, dtype=int)
array([[0, 0, 0],
       [0, 0, 0]])
```

### Unos

```
>>> np.ones((3, 4))  # también existe np.ones_like()
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

### Mismo valor

```
>>> np.full((3, 4), 7)  # también existe np.full_like()
array([[7, 7, 7, 7],
       [7, 7, 7, 7],
       [7, 7, 7, 7]])
```

### Matriz identidad

```
>>> np.eye(5)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

### Matriz diagonal

```
>>> np.diag([5, 4, 3, 2, 1])
array([[5, 0, 0, 0, 0],
       [0, 4, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 2, 0],
       [0, 0, 0, 0, 1]])
```

---

**Ejercicio**

Cree la siguiente matriz mediante código Python:

$$\text{diagonal} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 2 & \dots & 0 \\ \vdots & \vdots & 0 & \ddots & 0 \\ 0 & 0 & 0 & \dots & 49 \end{bmatrix}$$

Obtenga igualmente las siguientes características de cada uno de ellos: dimensión, tamaño, forma y tipo de sus elementos.

---

### Valores equiespaciados

A continuación veremos una serie de funciones para crear arrays con valores equiespaciados o en intervalos definidos.

#### Valores enteros equiespaciados

La función que usamos para este propósito es `np.arange()` cuyo comportamiento es totalmente análogo a la función «built-in» *range()*.

**Especificando límite superior:**

```
>>> np.arange(21)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20])
```

**Especificando límite inferior y superior:**

```
>>> np.arange(6, 60)
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
       23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
       40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
       57, 58, 59])
```

**Especificando límite inferior, superior y paso:**

```
>>> np.arange(6, 60, 3)
array([ 6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54,
       57])
```

Es posible especificar un **paso flotante** en la función `arange()`:

```
>>> np.arange(6, 16, .3)
array([ 6. ,  6.3,  6.6,  6.9,  7.2,  7.5,  7.8,  8.1,  8.4,  8.7,  9. ,
        9.3,  9.6,  9.9, 10.2, 10.5, 10.8, 11.1, 11.4, 11.7, 12. , 12.3,
       12.6, 12.9, 13.2, 13.5, 13.8, 14.1, 14.4, 14.7, 15. , 15.3, 15.6,
       15.9])
```

### Valores flotantes equiespaciados

La función que usamos para este propósito es `np.linspace()` cuyo comportamiento es «similar» a `np.arange()` pero para valores flotantes.

**Especificando límite inferior y superior:**

```
>>> np.linspace(6, 60)  # [6, 60] con 50 valores
array([ 6.        ,  7.10204082,  8.20408163,  9.30612245, 10.40816327,
       11.51020408, 12.6122449 , 13.71428571, 14.81632653, 15.91836735,
       17.02040816, 18.12244898, 19.2244898 , 20.32653061, 21.42857143,
       22.53061224, 23.63265306, 24.73469388, 25.83673469, 26.93877551,
       28.04081633, 29.14285714, 30.24489796, 31.34693878, 32.44897959,
       33.55102041, 34.65306122, 35.75510204, 36.85714286, 37.95918367,
       39.06122449, 40.16326531, 41.26530612, 42.36734694, 43.46938776,
       44.57142857, 45.67346939, 46.7755102 , 47.87755102, 48.97959184,
       50.08163265, 51.18367347, 52.28571429, 53.3877551 , 54.48979592,
       55.59183673, 56.69387755, 57.79591837, 58.89795918, 60.        ])
```

**Nota:** Por defecto `np.linspace()` genera 50 elementos.

**Especificando límite inferior, superior y total de elementos:**

```
>>> np.linspace(6, 60, 20)  # [6, 60] con 20 valores
array([ 6.        ,  8.84210526, 11.68421053, 14.52631579, 17.36842105,
       20.21052632, 23.05263158, 25.89473684, 28.73684211, 31.57894737,
       34.42105263, 37.26315789, 40.10526316, 42.94736842, 45.78947368,
       48.63157895, 51.47368421, 54.31578947, 57.15789474, 60.        ])
```

**Importante:** A diferencia de `np.arange()`, la función `np.linspace()` incluye «por defecto» el límte superior especificado.

**Especificando un intervalo abierto** $[a, b)$**:**

```
>>> np.linspace(6, 60, 20, endpoint=False)  # [6, 60) con 20 elementos
array([ 6. ,  8.7, 11.4, 14.1, 16.8, 19.5, 22.2, 24.9, 27.6, 30.3, 33. ,
       35.7, 38.4, 41.1, 43.8, 46.5, 49.2, 51.9, 54.6, 57.3])
```

### Valores aleatorios

A continuación veremos una serie de funciones para crear arrays con valores aleatorios y distribuciones de probabilidad.

#### Valores aleatorios enteros

**Valores aleatorios enteros en** $[a, b)$**:**

```
>>> np.random.randint(3, 30)  # escalar
4
```

```
>>> np.random.randint(3, 30, size=9)  # vector
array([29,  7,  8, 21, 27, 23, 29, 15, 28])
```

```
>>> np.random.randint(3, 30, size=(3, 3))  # matriz
array([[24,  4, 29],
       [10, 22, 27],
       [27,  7, 20]])
```

#### Valores aleatorios flotantes

Por simplicidad, en el resto de ejemplos vamos a obviar la salida *escalar* y *matriz*.

**Valores aleatorios flotantes en** $[0, 1)$**:**

```
>>> np.random.random(9)
array([0.53836208, 0.78315275, 0.6931254 , 0.97194325, 0.01523289,
       0.47692141, 0.27653964, 0.82297655, 0.70502383])
```

**Valores aleatorios flotantes en** $[a, b)$**:**

```
>>> np.random.uniform(1, 100, size=9)
array([17.00450378, 67.08416159, 56.99930273,  9.19685998, 35.27334323,
       97.34651516, 25.89283558, 53.59685476, 72.74943888])
```

### Distribuciones de probabilidad

**Distribución normal:** Ejemplo en el que generamos un millón de valores usando como parámetros de la distribución $\mu = 0, \sigma = 5$

```
>>> dist = np.random.normal(0, 5, size=1_000_000)

>>> dist[:20]
array([ 2.5290643 ,  1.46577658,  1.65170437, -1.36970819, -2.24547757,
        7.19905613, -4.4666239 , -1.05505116,  2.42351298, -4.45314272,
        1.13604077, -2.85054948,  4.34589478, -2.81235743, -0.8215143 ,
        0.57796411, -2.56594122, -7.14899388,  3.49197644,  1.80691996])

>>> dist.mean()
0.004992046432131982

>>> dist.std()
4.998583810032169
```

**Muestra aleatoria:** Ejemplo en el que generamos una muestra aleatoria de un millón de lanzamientos de una moneda:

```
>>> coins = np.random.choice(['head', 'tail'], size=1_000_000)

>>> coins
array(['tail', 'head', 'tail', ..., 'tail', 'head', 'tail'], dtype='<U4')

>>> sum(coins == 'head')
499874
>>> sum(coins == 'tail')
500126
```

**Muestra aleatoria con probabilidades no uniformes:** Ejemplo en el que generamos una muestra aleatoria de un millón de lanzamientos con un dado **«trucado»**:

```
>>> # La cara del "1" tiene un 50% de probabilidades de salir
>>> dices = np.random.choice(range(1, 7), size=1_000_000, p=[.5, .1, .1, .1, .1,
↪ .1])

>>> dices
array([6, 5, 4, ..., 1, 6, 1])

>>> sum(dices == 1)
500290
>>> sum(dices == 6)
99550
```

**Muestra aleatoria sin reemplazo:** Ejemplo en el que seleccionamos 5 principios aleatorios del *Zen de Python* sin reemplazo:

```
>>> import this
>>> import codecs

>>> zen = codecs.decode(this.s, 'rot-13').splitlines()[3:]  # https://bit.ly/
↪3xhsucQ

>>> np.random.choice(zen, size=5, replace=False)
array(['Unless explicitly silenced.',
       "Although that way may not be obvious at first unless you're Dutch.",
       'Sparse is better than dense.',
       'If the implementation is easy to explain, it may be a good idea.',
       'Complex is better than complicated.'], dtype='<U69')
```

**Ver también:**

Listado de distribuciones aleatorias que se pueden utilizar en NumPy.

---

**Ejercicio**

Cree:

- Una matriz de 20 filas y 5 columnas con valores flotantes equiespaciados en el intervalo cerrado $[1, 10]$.
- Un array unidimensional con 128 valores aleatorios de una distribución normal $\mu = 1, \sigma = 2$.
- Un array unidimensional con 15 valores aleatorios de una muestra *1, X, 2* donde la probabilidad de que gane el equipo local es del 50%, la probabilidad de que empaten es del 30% y la probabilidad de que gane el visitante es del 20%.

---

## Constantes

Numpy proporciona una serie de constantes predefinidas que facilitan su acceso y reutilización. Veamos algunas de ellas:

```
>>> import numpy as np

>>> np.Inf
inf

>>> np.nan
nan
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> np.e
2.718281828459045

>>> np.pi
3.141592653589793
```

### 9.2.3 Manipulando elementos

Los arrays multidimensionales de NumPy están indexados por unos ejes que establecen la forma en la que debemos acceder a sus elementos. Véase el siguiente diagrama:

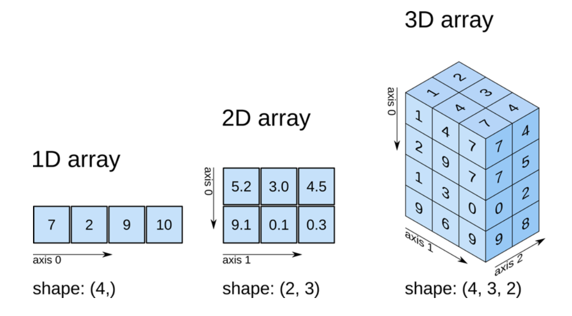



Figura 15: Esquema de ejes sobre los arrays de NumPy[3]

#### Arrays unidimensionales

#### Acceso a arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> values[2]
12
>>> values[-3]
13
```

[3] Imagen de Harriet Dashnow, Stéfan van der Walt y Juan Núñez-Iglesias en O'Reilly.

### Modificación a arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> values[0] = values[1] + values[5]

>>> values
array([26, 11, 12, 13, 14, 15])
```

### Borrado en arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> np.delete(values, 2)  # índice (como escalar)
array([10, 11, 13, 14, 15])

>>> np.delete(values, (2, 3, 4))  # índices (como tupla)
array([10, 11, 15])
```

**Nota:** La función `np.delete()` no es destructiva. Devuelve una copia modificada del array.

### Inserción en arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> np.append(values, 16)  # añade elementos al final
array([10, 11, 12, 13, 14, 15, 16])

>>> np.insert(values, 1, 101)  # añade elementos en una posición
array([ 10, 101,  11,  12,  13,  14,  15])
```

Para ambas funciones también es posible añadir varios elementos de una sola vez:

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> np.append(values, [16, 17, 18])
array([10, 11, 12, 13, 14, 15, 16, 17, 18])
```

**Nota:** La funciones `np.append()` y `np.insert()` no son destructivas. Devuelven una copia modificada del array.

### Arrays multidimensionales

Partimos del siguiente array bidimensional (matriz) para ejemplificar las distintas operaciones:

```
>>> values = np.arange(1, 13).reshape(3, 4)
>>> values
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

### Acceso a arrays multidimensionales

```
>>> values
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

**Acceso a elementos individuales::**

```
>>> values[0, 0]
1
>>> values[-1, -1]
12
>>> values[1, 2]
7
```

**Acceso a múltiples elementos::**

```
>>> values[[0, 2], [1, 2]]  # Elementos [0, 1] y [2, 2]
array([ 2, 11])
```

**Acceso a filas o columnas completas:**

```
>>> values[2]  # tercera fila
array([ 9, 10, 11, 12])

>>> values[:, 1]  # segunda columna
array([ 2,  6, 10])
```

**Acceso a zonas parciales del array:**

```
>>> values[0:2, 0:2]
array([[1, 2],
       [5, 6]])

>>> values[0:2, [1, 3]]
array([[2, 4],
       [6, 8]])
```

**Importante:** Todos estos accesos crean una copia (vista) del array original. Esto significa que, si modificamos un valor en el array copia, se ve reflejado en el original. Para evitar esta situación podemos usar la función `np.copy()` y desvincular la vista de su fuente.

### Modificación de arrays multidimensionales

```
>>> values
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> values[0, 0] = 100

>>> values[1] = [55, 66, 77, 88]

>>> values[:,2] = [30, 70, 110]

>>> values
array([[100,   2,  30,   4],
       [ 55,  66,  70,  88],
       [  9,  10, 110,  12]])
```

### Borrado en arrays multidimensionales

```
>>> values
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> np.delete(values, 0, axis=0)  # Borrado de la primera fila
array([[ 5,  6,  7,  8],
```

(continué en la próxima página)

(proviene de la página anterior)

```
       [ 9, 10, 11, 12]])

>>> np.delete(values, (1, 3), axis=1)  # Borrado de la segunda y cuarta columna
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

**Truco:** Tener en cuenta que `axis=0` hace referencia a **filas** y `axis=1` hace referencia a **columnas** tal y como describe el *diagrama* del comienzo de la sección.

### Inserción en arrays multidimensionales

**Añadir elementos al final del array:**

```
>>> values
array([[1, 2],
       [3, 4]])

>>> np.append(values, [[5, 6]], axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> np.append(values, [[5], [6]], axis=1)
array([[1, 2, 5],
       [3, 4, 6]])
```

**Insertar elementos en posiciones arbitrarias del array:**

```
>>> values
array([[1, 2],
       [3, 4]])

>>> np.insert(values, 0, [0, 0], axis=0)
array([[0, 0],
       [1, 2],
       [3, 4]])

>>> np.insert(values, 1, [0, 0], axis=1)
array([[1, 0, 2],
       [3, 0, 4]])
```

**Ejercicio**

Utilizando las operaciones de modificación, borrado e inserción, convierta la siguiente matriz:

$$\begin{bmatrix} 17 & 12 & 31 \\ 49 & 11 & 51 \\ 21 & 31 & 62 \\ 63 & 75 & 22 \end{bmatrix}$$

en esta:

$$\begin{bmatrix} 17 & 12 & 31 & 63 \\ 49 & 11 & 51 & 75 \\ 21 & 31 & 62 & 22 \end{bmatrix}$$

y luego en esta:

$$\begin{bmatrix} 17 & 12 & 31 & 63 \\ 49 & 49 & 49 & 63 \\ 21 & 31 & 62 & 63 \end{bmatrix}$$

### Apilando matrices

Hay veces que nos interesa combinar dos matrices (arrays en general). Una de los mecanismos que nos proporciona NumPy es el **apilado**.

**Apilado vertical:**

```
>>> m1 = np.random.randint(1, 100, size=(3, 2))
>>> m2 = np.random.randint(1, 100, size=(1, 2))

>>> m1
array([[68, 68],
       [10, 50],
       [87, 92]])
>>> m2
array([[63, 80]])

>>> np.vstack((m1, m2))
array([[68, 68],
       [10, 50],
       [87, 92],
       [63, 80]])
```

**Apilado horizontal:**

```
>>> m1 = np.random.randint(1, 100, size=(3, 2))
>>> m2 = np.random.randint(1, 100, size=(3, 1))

>>> m1
array([[51, 50],
      [52, 15],
      [14, 21]])
>>> m2
array([[18],
       [52],
       [ 1]])

>>> np.hstack((m1, m2))
array([[51, 50, 18],
       [52, 15, 52],
       [14, 21,  1]])
```

### Repitiendo elementos

**Repetición por ejes:** El parámetro de repetición indica el número de veces que repetimos el array completo por cada eje:

```
>>> values
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> np.tile(values, 3)  # x3 en columnas
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4],
       [5, 6, 5, 6, 5, 6]])

>>> np.tile(values, (2, 3))  # x2 en filas; x3 en columnas
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4],
       [5, 6, 5, 6, 5, 6],
       [1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4],
       [5, 6, 5, 6, 5, 6]])
```

**Repetición por elementos:** El parámetro de repetición indica el número de veces que repetimos cada elemento del array:

```
>>> values
array([[1, 2],
```

(continué en la próxima página)

(proviene de la página anterior)

```
       [3, 4],
       [5, 6]])

>>> np.repeat(values, 2)
array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6])

>>> np.repeat(values, 2, axis=0)  # x2 en filas
array([[1, 2],
       [1, 2],
       [3, 4],
       [3, 4],
       [5, 6],
       [5, 6]])

>>> np.repeat(values, 3, axis=1)  # x3 en columnas
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4],
       [5, 5, 5, 6, 6, 6]])
```

### Acceso por diagonal

Es bastante común acceder a elementos de una matriz (array en general) tomando como referencia su diagonal. Para ello, NumPy nos provee de ciertos mecanismos que veremos a continuación.

Para ejemplificarlo, partiremos del siguiente array:

```
>>> values
array([[73, 86, 90, 20],
       [96, 55, 15, 48],
       [38, 63, 96, 95],
       [13, 87, 32, 96]])
```

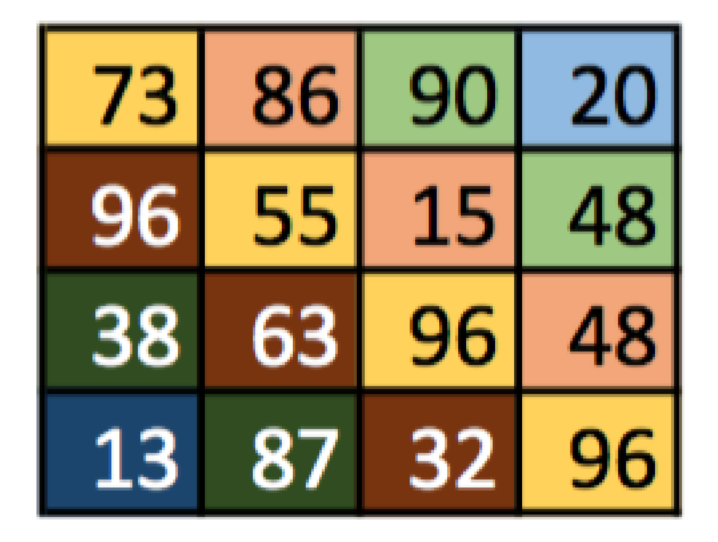
### Extracción de elementos por diagonal

La función `np.diag()` permite acceder a los elementos de un array especificando un parámetro `k` que indica la «distancia» con la diagonal principal:

Veamos cómo variando el parámetro `k` obtenemos distintos resultados:

```
>>> np.diag(values)  # k = 0
array([73, 55, 96, 96])
```

(continué en la próxima página)



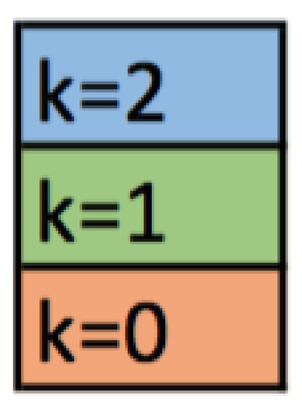










Figura 16: Acceso a elementos de un array por su diagonal

(proviene de la página anterior)

```
>>> for k in range(1, values.shape[0]):
...     print(f'k={k}', np.diag(values, k=k))
...
k=1 [86 15 95]
k=2 [90 48]
k=3 [20]

>>> for k in range(1, values.shape[0]):
...     print(f'k={-k}', np.diag(values, k=-k))
...
k=-1 [96 63 32]
k=-2 [38 87]
k=-3 [13]
```

### Modificación de elementos por diagonal

NumPy también provee un método `np.diag_indices()` que retorna los índices de los elementos de la diagonal principal, con lo que podemos modificar sus valores directamente:

```
>>> values
array([[73, 86, 90, 20],
       [96, 55, 15, 48],
       [38, 63, 96, 95],
       [13, 87, 32, 96]])

>>> di = np.diag_indices(values.shape[0])

>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))

>>> values[di] = 0

>>> values
array([[ 0, 86, 90, 20],
       [96,  0, 15, 48],
       [38, 63,  0, 95],
       [13, 87, 32,  0]])
```

**Truco:** Existen igualmente las funciones `np.triu_indices()` y `np.tril_indices()` para obtener los índices de la diagonal superior e inferior de una matriz.

### 9.2.4 Operaciones sobre arrays

#### Operaciones lógicas

#### Indexado booleano

El indexado booleano es una operación que permite conocer (a nivel de elemento) si un array cumple o no con una determinada condición:

```
>>> values
array([[60, 47, 34, 38],
       [43, 63, 37, 68],
       [58, 28, 31, 43],
       [32, 65, 32, 96]])

>>> values > 50  # indexado booleano
array([[ True, False, False, False],
       [False,  True, False,  True],
       [ True, False, False, False],
       [False,  True, False,  True]])

>>> values[values > 50]  # uso de máscara
array([60, 63, 68, 58, 65, 96])

>>> values[values > 50] = -1  # modificación de valores

>>> values
array([[-1, 47, 34, 38],
       [43, -1, 37, -1],
       [-1, 28, 31, 43],
       [32, -1, 32, -1]])
```

Las condiciones pueden ser más complejas e incorporar operadores lógicos `|` (or) y `&` (and):

```
>>> values
array([[60, 47, 34, 38],
       [43, 63, 37, 68],
       [58, 28, 31, 43],
       [32, 65, 32, 96]])

>>> (values < 25) | (values > 75)
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True]])
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> (values > 25) & (values < 75)
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True, False]])
```

**Consejo:** El uso de paréntesis es obligatorio si queremos mantener la precedencia y que funcione correctamente.

**Ejercicio**

Extraiga todos los números impares de la siguiente matriz:

$$\text{values} = \begin{bmatrix} 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 \end{bmatrix}$$

Si lo que nos interesa es **obtener los índices** del array que satisfacen una determinada condición, NumPy nos proporciona el método `where()` cuyo comportamiento se ejemplifica a continuación:

```
>>> values
array([[60, 47, 34, 38],
       [43, 63, 37, 68],
       [58, 28, 31, 43],
       [32, 65, 32, 96]])

>>> idx = np.where(values > 50)

>>> idx
(array([0, 1, 1, 2, 3, 3]), array([0, 1, 3, 0, 1, 3]))

>>> values[idx]
array([60, 63, 68, 58, 65, 96])
```

**Ejercicio**

Partiendo de una matriz de 10 filas y 10 columnas con valores aleatorios enteros en el intervalo $[0, 100]$, realice las operaciones necesarias para obtener una matriz de las mismas dimensiones donde:

- Todos los elementos de la diagonal sean 50.

- Los elementos mayores que 50 tengan valor 100.
- Los elementos menores que 50 tengan valor 0.

---

### Comparando arrays

Dados dos arrays podemos compararlos usando el operador `==` del mismo modo que con cualquier otro objeto en Python. La cuestión es que el resultado se evalúa a nivel de elemento:

```
>>> m1 = np.array([[1, 2], [3, 4]])
>>> m3 = np.array([[1, 2], [3, 4]])

>>> m1 == m2
array([[ True,  True],
       [ True,  True]])
```

Si queremos comparar arrays en su totalidad, podemos hacer uso de la siguiente función:

```
>>> np.array_equal(m1, m2)
True
```

### Operaciones de conjunto

Al igual que existen *operaciones sobre conjuntos* en Python, también podemos llevarlas a cabo sobre arrays en NumPy.

### Unión de arrays

$x \cup y$

```
>>> x
array([ 9,  4, 11,  3, 14,  5, 13, 12,  7, 14])
>>> y
array([17,  9, 19,  4, 18,  4,  7, 13, 11, 10])

>>> np.union1d(x, y)
array([ 3,  4,  5,  7,  9, 10, 11, 12, 13, 14, 17, 18, 19])
```

### Intersección de arrays

$x \cap y$

```
>>> x
array([ 9,  4, 11,  3, 14,  5, 13, 12,  7, 14])
>>> y
array([17,  9, 19,  4, 18,  4,  7, 13, 11, 10])

>>> np.intersect1d(x, y)
array([ 4,  7,  9, 11, 13])
```

### Diferencia de arrays

$x \setminus y$

```
>>> x
array([ 9,  4, 11,  3, 14,  5, 13, 12,  7, 14])
>>> y
array([17,  9, 19,  4, 18,  4,  7, 13, 11, 10])

>>> np.setdiff1d(x, y)
array([ 3,  5, 12, 14])
```

### Ordenación de arrays

En términos generales, existen dos formas de ordenar cualquier estructura de datos, una que modifica «in-situ» los valores (destructiva) y otra que devuelve «nuevos» valores (no destructiva). En el caso de NumPy también es así.

### Ordenación sobre arrays unidimensionales

```
>>> values
array([23, 24, 92, 88, 75, 68, 12, 91, 94, 24,  9, 21, 42,  3, 66])

>>> np.sort(values)  # no destructivo
array([ 3,  9, 12, 21, 23, 24, 24, 42, 66, 68, 75, 88, 91, 92, 94])

>>> values.sort()  # destructivo

>>> values
array([ 3,  9, 12, 21, 23, 24, 24, 42, 66, 68, 75, 88, 91, 92, 94])
```

### Ordenación sobre arrays multidimensionales

```
>>> values
array([[52, 23, 90, 46],
       [61, 63, 74, 59],
       [75,  5, 58, 70],
       [21,  7, 80, 52]])

>>> np.sort(values, axis=1)  # equivale a np.sort(values)
array([[23, 46, 52, 90],
       [59, 61, 63, 74],
       [ 5, 58, 70, 75],
       [ 7, 21, 52, 80]])

>>> np.sort(values, axis=0)
array([[21,  5, 58, 46],
       [52,  7, 74, 52],
       [61, 23, 80, 59],
       [75, 63, 90, 70]])
```

**Nota:** También existe `values.sort(axis=1)` y `values.sort(axis=0)` como métodos «destructivos» de ordenación.

### Contando valores

Otra de las herramientas útiles que proporciona NumPy es la posibilidad de contar el número de valores que existen en un array en base a ciertos criterios.

Para ejemplificarlo, partiremos de un array unidimensional con valores de una distribución aleatoria uniforme en el intervalo $[1, 10]$:

```
>>> randomized = np.random.randint(1, 11, size=1000)

>>> randomized
array([ 7,  9,  7,  8,  3,  7,  6,  4,  3,  9,  3,  1,  6,  7, 10,  4,  8,
        1,  3,  3,  8,  5,  4,  7,  5,  8,  8,  3, 10,  1,  7, 10,  3, 10,
        2,  9,  5,  1,  2,  4,  4, 10,  5, 10,  5,  2,  5,  2, 10,  3,  4,
...
```

**Valores únicos:**

```
>>> np.unique(randomized)
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

**Valores únicos (incluyendo frecuencias):**

```
>>> np.unique(randomized, return_counts=True)
(array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
array([101,  97, 117,  94, 101,  88,  94, 110,  93, 105]))
```

**Valores distintos de cero:**

```
>>> np.count_nonzero(randomized)
1000
```

**Valores distintos de cero (incluyendo condición):**

```
>>> np.count_nonzero(randomized > 5)
490
```

### Operaciones aritméticas

Una de las grandes ventajas del uso de arrays numéricos en NumPy es la posibilidad de trabajar con ellos como si fueran objetos «simples» pero sacando partido de la aritmética vectorial. Esto redunda en una mayor eficiencia y rapidez de cómputo.

### Operaciones aritméticas con mismas dimensiones

Cuando operamos entre arrays de las mismas dimensiones, las operaciones aritméticas se realizan elemento a elemento (ocupando misma posición) y el resultado, obviamente, tiene las mismas dimensiones:

```
>>> m1
array([[21, 86, 45],
       [31, 36, 78],
       [31, 64, 70]])

>>> m2
array([[58, 67, 17],
       [99, 53,  9],
       [92, 42, 75]])

>>> m1 + m2
array([[ 79, 153,  62],
       [130,  89,  87],
       [123, 106, 145]])

>>> m1 - m2
```

(continué en la próxima página)

(proviene de la página anterior)

```
array([[-37,  19,  28],
       [-68, -17,  69],
       [-61,  22,  -5]])

>>> m1 * m2
array([[1218, 5762,  765],
       [3069, 1908,  702],
       [2852, 2688, 5250]])

>>> m1 / m2  # división flotante
array([[0.36206897, 1.28358209, 2.64705882],
       [0.31313131, 0.67924528, 8.66666667],
       [0.33695652, 1.52380952, 0.93333333]])

>>> m1 // m2  # división entera
array([[0, 1, 2],
       [0, 0, 8],
       [0, 1, 0]])
```

### Operaciones aritméticas con distintas dimensiones

Cuando operamos entre arrays con dimensiones diferentes, siempre y cuando se cumplan ciertas restricciones en tamaños de filas y/o columnas, lo que se produce es un «broadcasting» (o difusión) de los valores.

**Suma con array «fila»:**

```
>>> m
array([[9, 8, 1],
       [7, 6, 7]])

>>> v
array([[2, 3, 6]])

>>> m + v  # broadcasting
array([[11, 11,  7],
       [ 9,  9, 13]])
```

**Suma con array «columna»:**

```
>>> m
array([[9, 8, 1],
       [7, 6, 7]])
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> v
array([[1],
       [6]])

>>> m + v  # broadcasting
array([[10,  9,  2],
       [13, 12, 13]])
```

**Advertencia:** En el caso de que no coincidan dimensiones de filas y/o columnas, NumPy no podrá ejecutar la operación y obtendremos un error `ValueError: operands could not be broadcast together with shapes`.

### Operaciones entre arrays y escalares

Al igual que ocurría en los casos anteriores, si operamos con un array y un escalar, éste último será difundido para abarcar el tamaño del array:

```
>>> m
array([[9, 8, 1],
       [7, 6, 7]])

>>> m + 5
array([[14, 13,  6],
       [12, 11, 12]])

>>> m - 5
array([[ 4,  3, -4],
       [ 2,  1,  2]])

>>> m * 5
array([[45, 40,  5],
       [35, 30, 35]])

>>> m / 5
array([[1.8, 1.6, 0.2],
       [1.4, 1.2, 1.4]])

>>> m // 5
array([[1, 1, 0],
       [1, 1, 1]])

>>> m ** 5
```

(continué en la próxima página)

(proviene de la página anterior)

```
array([[59049, 32768,     1],
       [16807,  7776, 16807]])
```

### Operaciones unarias

Existen multitud de operaciones sobre un único array. A continuación veremos algunas de las más utilizas en NumPy.

### Funciones universales

Las funciones universales «ufunc» son funciones que operan sobre arrays **elemento a elemento**. Existen muchas funciones universales definidas en Numpy, parte de ellas operan sobre dos arrays y parte sobre un único array.

Un ejemplo de algunas de estas funciones:

```
>>> values
array([[48.32172375, 24.89651106, 77.49724241],
       [77.81874191, 22.54051494, 65.11282444],
       [ 5.54960482, 59.06720303, 62.52817198]])

>>> np.sqrt(values)
array([[6.95138287, 4.98964037, 8.80325181],
       [8.82149318, 4.74768522, 8.06925179],
       [2.35575992, 7.68551905, 7.9074757 ]])

>>> np.sin(values)
array([[-0.93125201, -0.23403917,  0.86370435],
       [ 0.66019205, -0.52214693,  0.75824777],
       [-0.66953344,  0.58352079, -0.29903488]])

>>> np.ceil(values)
array([[49., 25., 78.],
       [78., 23., 66.],
       [ 6., 60., 63.]])

>>> np.floor(values)
array([[48., 24., 77.],
       [77., 22., 65.],
       [ 5., 59., 62.]])

>>> np.log(values)
array([[3.87788123, 3.21472768, 4.35024235],
```

(continué en la próxima página)

(proviene de la página anterior)

```
       [4.3543823 , 3.11531435, 4.17612153],
       [1.71372672, 4.07867583, 4.13561721]])
```

## Reduciendo el resultado

NumPy nos permite aplicar cualquier función sobre un array **reduciendo** el resultado por alguno de sus ejes. Esto abre una amplia gama de posibilidades.

A modo de ilustración, veamos un par de ejemplos con la suma y el producto:

```
>>> values
array([[8, 2, 7],
       [2, 0, 6],
       [6, 3, 4]])

>>> np.sum(values, axis=0)  # suma por columnas
array([16,  5, 17])

>>> np.sum(values, axis=1)  # suma por filas
array([17,  8, 13])

>>> np.prod(values, axis=0)  # producto por columnas
array([ 96,   0, 168])

>>> np.prod(values, axis=1)  # producto por filas
array([112,   0,  72])
```

### Ejercicio

Compruebe que, para $\theta = 2\pi$ (*radianes*) y $k = 20$ se cumple la siguiente igualdad del *producto infinito de Euler*:

$$\cos\left(\frac{\theta}{2}\right) \cdot \cos\left(\frac{\theta}{4}\right) \cdot \cos\left(\frac{\theta}{8}\right) \cdots = \prod_{i=1}^{k} \cos\left(\frac{\theta}{2^i}\right) \approx \frac{\sin(\theta)}{\theta}$$

### Funciones estadísticas

NumPy proporciona una gran cantidad de funciones estadísticas que pueden ser aplicadas sobre arrays.

Veamos algunas de ellas:

```
>>> dist
array([[-6.79006504, -0.01579498, -0.29182173,  0.3298951 , -5.30598975],
       [ 3.10720923, -4.09625791, -7.60624152,  2.3454259 ,  9.23399023],
       [-7.4394269 , -9.68427195,  3.04248586, -5.9843767 ,  1.536578  ],
       [ 3.33953286, -8.41584411, -9.530274  , -2.42827813, -7.34843663],
       [ 7.1508544 ,  5.51727548, -3.20216834, -5.00154367, -7.15715252]])

>>> np.mean(dist)
-2.1877878715377777

>>> np.std(dist)
5.393254994089515

>>> np.median(dist)
-3.2021683412383295
```

### Máximos y mínimos

Una de las operaciones más comunes en el manejo de datos es encontrar máximos o mínimos. Para ello, disponemos de las típicas funciones con las ventajas del uso de arrays multidimensionales:

```
>>> values
array([[66, 54, 33, 15, 58],
       [55, 46, 39, 16, 38],
       [73, 75, 79, 25, 83],
       [81, 30, 22, 32,  8],
       [92, 25, 82, 10, 90]])

>>> np.min(values)
8

>>> np.min(values, axis=0)
array([55, 25, 22, 10,  8])
>>> np.min(values, axis=1)
array([15, 16, 25,  8, 10])

>>> np.max(values)
```

(continué en la próxima página)

(proviene de la página anterior)

```
92

>>> np.max(values, axis=0)
array([92, 75, 82, 32, 90])
>>> np.max(values, axis=1)
array([66, 55, 83, 81, 92])
```

Si lo que interesa es obtener los **índices** de aquellos elementos con valores máximos o mínimos, podemos hacer uso de las funciones `argmax()` y `argmin()` respectivamente.

Veamos un ejemplo donde obtenemos los valores máximos por columnas (mediante sus índices):

```
>>> values
array([[66, 54, 33, 15, 58],
       [55, 46, 39, 16, 38],
       [73, 75, 79, 25, 83],
       [81, 30, 22, 32,  8],
       [92, 25, 82, 10, 90]])

>>> idx = np.argmax(values, axis=0)

>>> idx
array([4, 2, 4, 3, 4])

>>> values[idx, range(values.shape[1])]
array([92, 75, 82, 32, 90])
```

### Vectorizando funciones

Una de las ventajas de trabajar con arrays numéricos en NumPy es sacar provecho de la optimización que se produce a nivel de la propia estructura de datos. En el caso de que queramos implementar una función propia para realizar una determinada acción, sería deseable seguir aprovechando esa característica.

Veamos un ejemplo en el que queremos realizar el siguiente cálculo entre dos matrices $A$ y $B$:

$$A_{ij} * B_{ij} = \begin{cases} A_{ij} + B_{ij} & \text{, si } A_{ij} > B_{ij} \\ A_{ij} - B_{ij} & \text{, si } A_{ij} < B_{ij} \\ 0 & \text{, e.o.c.} \end{cases}$$

Esta función, definida en Python, quedaría tal que así:

```
>>> def customf(a, b):
...     if a > b:
...         return a + b
...     elif a < b:
...         return a - b
...     else:
...         return 0
...
```

Las dos matrices de partida tienen 9M de valores aleatorios entre -100 y 100:

```
>>> A = np.random.randint(-100, 100, size=(3000, 3000))
>>> B = np.random.randint(-100, 100, size=(3000, 3000))
```

Una primera aproximación para aplicar esta función a cada elemento de las matrices de entrada sería la siguiente:

```
>>> result = np.zeros_like(A)

>>> %%timeit
... for i in range(A.shape[0]):
...     for j in range(A.shape[1]):
...         result[i, j] = customf(A[i, j], B[i, j])
...
3 s ± 23.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

### Mejorando rendimiento con funciones vectorizadas

Con un pequeño detalle podemos mejorar el rendimiento de la función que hemos diseñado anteriormente. Se trata de decorarla con `np.vectorize` con lo que estamos otorgándole un comportamiento distinto y enfocado al procesamiento de arrays numéricos:

```
>>> @np.vectorize
... def customf(a, b):
...     if a > b:
...         return a + b
...     elif a < b:
...         return a - b
...     else:
...         return 0
...
```

Dado que ahora ya se trata de una **función vectorizada** podemos aplicarla directamente a las matrices de entrada (aprovechamos para medir su tiempo de ejecución):

```
>>> %timeit customf(A, B)
1.29 s ± 7.33 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Hemos obtenido una mejora de `2.32x` con respecto al uso de funciones simples.

**Truco:** La mejora de rendimiento se aprecia más claramente a medida que los tamaños de las matrices (arrays) de entrada son mayores.

**Consejo:** El uso de *funciones lambda* puede ser muy útil en vectorización: `np.vectorize(lambda a, b: return a + b)`.

**Ejercicio**

1. Cree dos matrices cuadradas de 20x20 con valores aleatorios flotantes uniformes en el intervalo $[0, 1000)$
2. Vectorice una función que devuelva la media (elemento a elemento) entre las dos matrices.
3. Realice la misma operación que en 2) pero usando suma de matrices y división por escalar.
4. Compute los tiempos de ejecución de 2) y 3)

### 9.2.5 Álgebra lineal

NumPy tiene una sección dedicada al álgebra lineal cuyas funciones pueden resultar muy interesantes según el contexto en el que estemos trabajando.

#### Producto de matrices

Si bien hemos hablado del producto de arrays elemento a elemento, NumPy nos permite hacer la multiplicación clásica de matrices:

```
>>> m1
array([[1, 8, 4],
       [8, 7, 1],
       [1, 3, 8]])

>>> m2
```

(continué en la próxima página)

(proviene de la página anterior)

```
array([[1, 5, 7],
       [9, 4, 2],
       [1, 4, 2]])

>>> np.dot(m1, m2)
array([[77, 53, 31],
       [72, 72, 72],
       [36, 49, 29]])
```

En Python 3.5 se introdujo el operador `@` que permitía implementar el *método especial* `__matmul__()` de multiplicación de matrices. NumPy lo ha desarrollado y simplifica la multiplicación de matrices de la siguiente manera:

```
>>> m1 @ m2
array([[77, 53, 31],
       [72, 72, 72],
       [36, 49, 29]])
```

**Ejercicio**

Compruebe que la matriz $\begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix}$ satisface la ecuación matricial: $X^2 - 6X - I = 0$ donde $I$ es la matriz identidad de orden 2.

### Determinante de una matriz

El cálculo del determinante es una operación muy utilizada en álgebra lineal. Lo podemos realizar en NumPy de la siguiente manera:

```
>>> m
array([[4, 1, 6],
       [4, 8, 8],
       [2, 1, 7]])

>>> np.linalg.det(m)
108.00000000000003
```

### Inversa de una matriz

La inversa de una matriz se calcula de la siguiente manera:

```
>>> m
array([[4, 1, 6],
       [4, 8, 8],
       [2, 1, 7]])

>>> m_inv = np.linalg.inv(m)

>>> m_inv
array([[ 0.44444444, -0.00925926, -0.37037037],
       [-0.11111111,  0.14814815, -0.07407407],
       [-0.11111111, -0.01851852,  0.25925926]])
```

Una propiedad de la matriz inversa es que si la multiplicamos por la matriz de partida obtenemos la matriz identidad. Vemos que se cumple $\mathcal{A} \cdot \mathcal{A}^{-1} = \mathcal{I}$:

```
>>> np.dot(m, m_inv)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

### Traspuesta de una matriz

La traspuesta de una matriz $\mathcal{A}$ se denota por: $(\mathcal{A}^t)_{ij} = \mathcal{A}_{ji},\ 1 \leq i \leq n,\ 1 \leq j \leq m$, pero básicamente consiste en intercambiar filas por columnas.

Aún más fácil es computar la traspuesta de una matriz con NumPy:

```
>>> m
array([[1, 2, 3],
       [4, 5, 6]])

>>> m.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

---

**Ejercicio**

Dadas las matrices:

$$A = \begin{bmatrix} 1 & -2 & 1 \\ 3 & 0 & 1 \end{bmatrix};\ B = \begin{bmatrix} 4 & 0 & -1 \\ -2 & 1 & 0 \end{bmatrix}$$

, compruebe que se cumplen las siguientes igualdades:

- $(A+B)^t = A^t + B^t$
- $(3A)^t = 3A^t$

### Elevar matriz a potencia

En el mundo del álgebra lineal es muy frecuente recurrir a la exponenciación de matrices a través de su producto clásico. En este sentido, NumPy nos proporciona una función para computarlo:

```
>>> m
array([[4, 1, 6],
       [4, 8, 8],
       [2, 1, 7]])

>>> np.linalg.matrix_power(m, 3)  # más eficiente que np.dot(m, np.dot(m, m))
array([[ 348,  250,  854],
       [ 848,  816, 2000],
       [ 310,  231,  775]])
```

**Ejercicio**

Dada la matriz $A = \begin{bmatrix} 4 & 5 & -1 \\ -3 & -4 & 1 \\ -3 & -4 & 0 \end{bmatrix}$ calcule: $A^2, A^3, \ldots, A^{128}$

¿Nota algo especial en los resultados?

### Sistemas de ecuaciones lineales

NumPy también nos permite resolver sistemas de ecuaciones lineales. Para ello debemos modelar nuestro sistema a través de arrays.

Veamos un ejemplo en el que queremos resolver el siguiente sistema de ecuaciones lineales:

$$\begin{cases} x_1 + 2x_3 = 1 \\ x_1 - x_2 = -2 \\ x_2 + x_3 = -1 \end{cases} \implies \begin{pmatrix} 1 & 0 & 2 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix} \implies \mathcal{A}\mathcal{X} = \mathcal{B}$$

Podemos almacenar las matrices de coeficientes $\mathcal{A}$ y $\mathcal{B}$ de la siguiente manera:

```
>>> A = np.array([[1, 0, 2], [1, -1, 0], [0, 1, 1]])
>>> B = np.array([1, -2, -1]).reshape(-1, 1)

>>> A
array([[ 1,  0,  2],
       [ 1, -1,  0],
       [ 0,  1,  1]])
>>> B
array([[ 1],
       [-2],
       [-1]])
```

La solución al sistema viene dada por la siguiente función:

```
>>> np.linalg.solve(A, B)
array([[-7.],
       [-5.],
       [ 4.]])
```

La solución del sistema debe ser la misma que si obtenemos $\mathcal{X} = \mathcal{A}^{-1} \cdot B$:

```
>>> np.dot(np.linalg.inv(A), B)
array([[-7.],
       [-5.],
       [ 4.]])
```

---

**Ejercicio**

Resuelva el siguiente sistema de ecuaciones lineales:

$$\begin{cases} 3x + 4y - z = 8 \\ 5x - 2y + z = 4 \\ 2x - 2y + z = 1 \end{cases}$$

---

## 9.3 pandas



pandas es un paquete open-source que nos proporciona una forma sencilla y potente de trabajar con estructuras de datos a través de múltiples herramientas para su análisis.[1]

```
$ pip install pandas
```

La forma más común de importar esta librería es usar el alias `pd`:

```
>>> import pandas as pd
```

Si bien en *Numpy* la estructura de datos fundamental es el `ndarray`, en pandas existen dos estructuras de datos sobre las que giran todas las operaciones:

- Series.
- Dataframes.

[1] Foto original de portada por Sid Balachandran en Unsplash.

## 9.3.1 Series

Podríamos pensar en una **serie** como un *ndarray* en el que cada valor tiene asignado una etiqueta (índice) y además admite un título (nombre).

### Creación de una serie

Veamos varios ejemplos de creación de la serie $[1, 2, 3]$.

**Creación de series usando listas:**

```
>>> pd.Series([1, 2, 3])
0    1
1    2
2    3
dtype: int64
```

---

**Nota:** El índice por defecto se crea con números enteros positivos empezando desde 0.

---

**Especificando un índice personalizado (etiqueta a cada valor):**

```
>>> pd.Series(range(1, 4), index=['a', 'b', 'c'])
a    1
b    2
c    3
dtype: int64
```

**Especificando un diccionario con etiquetas y valores:**

```
>>> items = {'a': 1, 'b': 2, 'c': 3}

>>> pd.Series(items)
a    1
b    2
c    3
dtype: int64
```

Todas las series que hemos visto hasta ahora no tienen asignado ningún nombre. Lo podemos hacer usando el parámetro `name` en la creación de la serie:

```
>>> pd.Series([1, 2, 3], name='integers')
0    1
1    2
```

(continué en la próxima página)

(proviene de la página anterior)

```
2    3
Name: integers, dtype: int64
```

**Ejercicio**

Cree una serie de pandas con valores enteros en el intervalo $[1, 26]$ y etiquetas `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Busque una manera programática (no manual) de hacerlo (recuerde el *módulo string*).

### Atributos de una serie

Las series en pandas contienen gran cantidad de atributos. A continuación destacaremos algunos de ellos.

Trabajaremos con datos que contienen el número de empleados/as de diferentes empresas tecnológicas[2]:

```
>>> data
{'Apple': 147000,
'Samsung': 267937,
'Google': 135301,
'Microsoft': 163000,
'Huawei': 197000,
'Dell': 158000,
'Facebook': 58604,
'Foxconn': 878429,
'Sony': 109700}

>>> employees = pd.Series(data, name='Tech Employees')
```

**Índice de la serie:**

```
>>> employees.index
Index(['Apple', 'Samsung', 'Google', 'Microsoft', 'Huawei', 'Dell', 'Facebook',
    'Foxconn', 'Sony'],
    dtype='object')
```

**Valores de la serie:**

```
>>> employees.values
array([147000, 267937, 135301, 163000, 197000, 158000,  58604, 878429,
    109700])
```

[2] Fuente: Wikipedia.

**Tipo de la serie:**

```
>>> employees.dtype
dtype('int64')
```

**Nombre de la serie:**

```
>>> employees.name
'Tech Employees'
```

**Memoria ocupada por la serie:**

```
>>> employees.nbytes
72
```

**Número de registros de la serie:**

```
>>> employees.size
9
```

### Selección de registros

La selección de los datos se puede realizar desde múltiples aproximaciones. A continuación veremos las posiblidades que nos ofrece pandas para seleccionar/filtrar los registros de una serie.

```
>>> employees
Apple        147000
Samsung      267937
Google       135301
Microsoft    163000
Huawei       197000
Dell         158000
Facebook      58604
Foxconn      878429
Sony         109700
Name: Tech Employees, dtype: int64
```

### Selección usando indexado numérico

Para acceder a los registros por su posición (índice numérico) basta usar corchetes como ya se ha visto en cualquier secuencia:

```
>>> employees[0]
147000

>>> employees[-1]
109700

>>> employees[2:5]
Google       135301
Microsoft    163000
Huawei       197000
Name: Tech Employees, dtype: int64

>>> employees[1:6:2]
Samsung      267937
Microsoft    163000
Dell         158000
Name: Tech Employees, dtype: int64
```

El atributo `iloc` es un alias (algo más expresivo) que permite realizar las mismas operaciones de indexado (con corchetes) que hemos visto anteriormente:

```
>>> employees.iloc[1:6:2]
Samsung      267937
Microsoft    163000
Dell         158000
Name: Tech Employees, dtype: int64
```

**Truco:** Python, y en este caso pandas, se dicen «0-index» porque sus índices (posiciones) comienzan en cero.

### Selección usando etiquetas

En el caso de aquellas series que dispongan de un índice con etiquetas, podemos acceder a sus registros utilizando las mismas:

```
>>> employees['Apple']  # equivalente a employees.Apple
147000
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> employees['Apple':'Huawei']
Apple        147000
Samsung      267937
Google       135301
Microsoft    163000
Huawei       197000
Name: Tech Employees, dtype: int64

>>> employees['Apple':'Huawei':2]
Apple     147000
Google    135301
Huawei    197000
Name: Tech Employees, dtype: int64
```

El atributo `loc` es un alias (algo más expresivo) que permite realizar las mismas operaciones de indexado (con corchetes) que hemos visto anteriormente:

```
>>> employees.loc['Apple':'Huawei':2]
Apple     147000
Google    135301
Huawei    197000
Name: Tech Employees, dtype: int64
```

### Fragmentos de comienzo y fin

A nivel exploratorio, es bastante cómodo acceder a una porción inicial (o final) de los datos que manejamos. Esto se puede hacer de forma muy sencilla con series:

```
>>> employees.head(3)
Apple      147000
Samsung    267937
Google     135301
Name: Tech Employees, dtype: int64

>>> employees.tail(3)
Facebook     58604
Foxconn     878429
Sony        109700
Name: Tech Employees, dtype: int64
```

### Operaciones con series

Si tenemos en cuenta que una serie contiene valores en formato `ndarray` podemos concluir que las *operaciones sobre arrays* son aplicables al caso de las series. Veamos algunos ejemplos de operaciones que podemos aplicar sobre series.

#### Operaciones lógicas

Supongamos que queremos filtrar aquellas empresas que tengan más de 200000 trabajadores/as:

```
>>> employees > 200_000
Apple        False
Samsung       True
Google       False
Microsoft    False
Huawei       False
Dell         False
Facebook     False
Foxconn       True
Sony         False
Name: Tech Employees, dtype: bool
```

Hemos obtenido una serie «booleana». Si queremos aplicar esta «máscara», podemos hacerlo con indexado:

```
>>> employees[employees > 200_000]  # empresas con más de 200K trabajadores/as
Samsung    267937
Foxconn    878429
Name: Tech Employees, dtype: int64
```

#### Ordenación

**Ordenación de una serie por sus valores:**

```
>>> employees.sort_values()
Facebook      58604
Sony         109700
Google       135301
Apple        147000
Dell         158000
Microsoft    163000
Huawei       197000
```

(continué en la próxima página)

(proviene de la página anterior)

```
Samsung      267937
Foxconn      878429
Name: Tech Employees, dtype: int64
```

**Ordenación de una serie por su índice:**

```
>>> employees.sort_index()
Apple        147000
Dell         158000
Facebook      58604
Foxconn      878429
Google       135301
Huawei       197000
Microsoft    163000
Samsung      267937
Sony         109700
Name: Tech Employees, dtype: int64
```

**Truco:** Ambos métodos admiten el parámetro `ascending` para indicar si la ordenación es ascendente (`True`) o descendente (`False`); y también admiten el parámetro `inplace` para indicar si se quiere modificar los valores de la serie (`True`) o devolver una nueva ya ordenada (`False`).

## Contando valores

Si queremos obtener una «tabla de frecuencias» podemos contar los valores que existen en nuestra serie:

```
>>> marks = pd.Series([5, 5, 3, 6, 5, 2, 8, 3, 8, 7, 6])

>>> marks.value_counts()
5    3
3    2
6    2
8    2
2    1
7    1
dtype: int64
```

Vinculado con el caso anterior, podemos obtener el número de valores únicos en la serie:

```
>>> marks.nunique()
6
```

El método `count()` devuelve el número de valores «no nulos» que contiene la serie:

```
>>> marks.count()  # en este caso es equivalente a marks.size
11
```

### Operaciones aritméticas

#### Operaciones entre series y escalares

Podemos operar entre series y escalares sin ningún tipo de problema:

```
>>> employees / 1000
Apple         147.000
Samsung       267.937
Google        135.301
Microsoft     163.000
Huawei        197.000
Dell          158.000
Facebook       58.604
Foxconn       878.429
Sony          109.700
Name: Tech Employees, dtype: float64
```

#### Operaciones entre series

Para el caso de operaciones entre series, vamos a ejemplificarlo con las dos siguientes[3]:

```
>>> employees
Apple         147000
Samsung       267937
Google        135301
Microsoft     163000
Huawei        197000
Dell          158000
Facebook       58604
Foxconn       878429
Sony          109700
Name: Tech Employees, dtype: int64
```

(continué en la próxima página)

[3] Los datos de ingresos («revenues») están en billones (americanos) de dólares.

(proviene de la página anterior)

```
>>> revenues
Apple        274515
Samsung      200734
Google       182527
Microsoft    143015
Huawei       129184
Dell          92224
Facebook      85965
Foxconn      181945
Sony          84893
Name: Tech Revenues, dtype: int64
```

Supongamos que queremos calcular la ratio de ingresos por trabajador/a:

```
>>> revenues / employees
Apple        1.867449
Samsung      0.749184
Google       1.349044
Microsoft    0.877393
Huawei       0.655756
Dell         0.583696
Facebook     1.466879
Foxconn      0.207125
Sony         0.773865
dtype: float64
```

**Truco:** Tener en cuenta que las operaciones se realizan entre registros que tienen el mismo índice (etiqueta).

### Funciones estadísticas

Existen multitud de funciones estadísticas que podemos aplicar a una serie. Dependiendo del tipo de dato con el que estamos trabajando, serán más útiles unas que otras. Veamos dos funciones a modo de ejemplo:

```
>>> employees.mean()
234996.77777777778

>>> employees.std()
248027.7840619765
```

## Máximos y mínimos

El abanico de posibilidades es muy amplio en cuanto a la búsqueda de valores máximos y mínimos en una serie. Veamos lo que nos ofrece pandas a este respecto.

**Obtener valor mínimo/máximo de una serie:**

```
>>> employees.min()
58604
>>> employees.max()
878429
```

**Posición (índice) del valor mínimo/máximo de una serie:**

```
>>> employees.argmin()  # employees[6] = 58604
6
>>> employees.argmax()  # employees[7] = 878429
7
```

**Etiqueta (índice) del valor mínimo/máximo de una serie:**

```
>>> employees.idxmin()
'Facebook'
>>> employees.idxmax()
'Foxconn'
```

**Obtener los $n$ valores menores/mayores de una serie:**

```
>>> employees.nsmallest(3)
Facebook      58604
Sony         109700
Google       135301
Name: Tech Employees, dtype: int64

>>> employees.nlargest(3)
Foxconn    878429
Samsung    267937
Huawei     197000
Name: Tech Employees, dtype: int64
```

## Exportación de series

Suele ser bastante habitual intercambiar datos en distintos formatos (y aplicaciones). Para ello, pandas nos permite exportar una serie a multitud de formatos. Veamos algunos de ellos:

**Exportación de serie a lista:**

```
>>> employees.to_list()
[147000, 267937, 135301, 163000, 197000, 158000, 58604, 878429, 109700]
```

**Exportación de serie a diccionario:**

```
>>> employees.to_dict()
{'Apple': 147000,
'Samsung': 267937,
'Google': 135301,
'Microsoft': 163000,
'Huawei': 197000,
'Dell': 158000,
'Facebook': 58604,
'Foxconn': 878429,
'Sony': 109700}
```

**Exportación de serie a csv:**

```
>>> employees.to_csv()
',Tech Employees\nApple,147000\nSamsung,267937\nGoogle,135301\nMicrosoft,163000\
↪nHuawei,197000\nDell,158000\nFacebook,58604\nFoxconn,878429\nSony,109700\n'
```

**Exportación de serie a json:**

```
>>> employees.to_json()
'{"Apple":147000,"Samsung":267937,"Google":135301,"Microsoft":163000,"Huawei
↪":197000,"Dell":158000,"Facebook":58604,"Foxconn":878429,"Sony":109700}'
```

**Exportación de serie a `pandas.DataFrame`:**

```
>>> employees.to_frame()
           Tech Employees
Apple              147000
Samsung            267937
Google             135301
Microsoft          163000
Huawei             197000
Dell               158000
Facebook            58604
Foxconn            878429
Sony               109700
```

Y muchos otros como: `to_clipboard()`, `to_numpy()`, `to_pickle()`, `to_string()`, `to_xarray()`, `to_excel()`, `to_hdf()`, `to_latex()`, `to_markdown()`, `to_period()`, `to_sql()` o `to_timestamp()`.

## 9.3.2 DataFrames

Un DataFrame es una estructura tabular compuesta por series. Se trata del tipo de datos fundamental en pandas y sobre el que giran la mayoría de operaciones que podemos realizar.

SERIES

| | employees |
|---|---|
| Apple | 147000 |
| Samsung | 267937 |
| Google | 135301 |
| Microsoft | 163000 |
| Huawei | 197000 |
| Dell | 158000 |
| Facebook | 58604 |
| Foxconn | 878429 |
| Sony | 109700 |

\+

SERIES

| | revenues |
|---|---|
| Apple | 274515 |
| Samsung | 200734 |
| Google | 182527 |
| Microsoft | 143015 |
| Huawei | 129184 |
| Dell | 92224 |
| Facebook | 85965 |
| Foxconn | 181945 |
| Sony | 84893 |

=

DATAFRAME

| | employees | revenues |
|---|---|---|
| Apple | 147000 | 274515 |
| Samsung | 267937 | 200734 |
| Google | 135301 | 182527 |
| Microsoft | 163000 | 143015 |
| Huawei | 197000 | 129184 |
| Dell | 158000 | 92224 |
| Facebook | 58604 | 85965 |
| Foxconn | 878429 | 181945 |
| Sony | 109700 | 84893 |

Figura 17: Estructura de un DataFrame a partir de Series

### Creación de un DataFrame

Existen múltiples formas de crear un DataFrame en pandas. Veamos algunas de ellas.

### DataFrame desde diccionario de listas

Cada elemento del diccionario se convierte en una **columna**, donde su clave es el nombre y sus valores se despliegan en «vertical»:

```
>>> data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

>>> pd.DataFrame(data)
   A  B
0  1  4
1  2  5
2  3  6
```

### DataFrame desde lista de diccionarios

Cada elemento de la lista se convierte en una **fila**. Las claves de cada diccionario serán los nombres de las columnas y sus valores se despliegan en «horizontal»:

```
>>> data = [{'A': 1, 'B': 2, 'C': 3}, {'A': 4, 'B': 5, 'C': 6}]

>>> pd.DataFrame(data)
   A  B  C
0  1  2  3
1  4  5  6
```

### DataFrame desde lista de listas

Cada elemento de la lista se convierte en una **fila** y sus valores se despliegan en «horizontal». Los nombres de las columnas deben pasarse como parámetro opcional:

```
>>> data = [[1, 2], [3, 4], [5, 6]]

>>> pd.DataFrame(data, columns=['A', 'B'])
   A  B
0  1  2
1  3  4
2  5  6
```

### DataFrame desde series

```
>>> employees
Apple        147000
Samsung      267937
Google       135301
Microsoft    163000
Huawei       197000
Dell         158000
Facebook      58604
Foxconn      878429
Sony         109700
Name: Tech Employees, dtype: int64

>>> revenues
Apple        274515
Samsung      200734
```

(continué en la próxima página)

(proviene de la página anterior)

```
Google       182527
Microsoft    143015
Huawei       129184
Dell          92224
Facebook      85965
Foxconn      181945
Sony          84893
Name: Tech Revenues, dtype: int64

>>> pd.DataFrame({'employees': employees, 'revenues': revenues})
           employees  revenues
Apple         147000    274515
Samsung       267937    200734
Google        135301    182527
Microsoft     163000    143015
Huawei        197000    129184
Dell          158000     92224
Facebook       58604     85965
Foxconn       878429    181945
Sony          109700     84893
```

---

**Ejercicio**

Cree el siguiente DataFrame en Pandas[5]:

[5] Datos extraídos de Wikipedia.

| Island | Population | Area | Province |
|---|---|---|---|
| Gran Canaria | 855521 | 1560.1 | LPGC |
| Tenerife | 928604 | 2034.38 | SCTF |
| La Palma | 83458 | 708.32 | SCTF |
| Lanzarote | 155812 | 845.94 | LPGC |
| La Gomera | 21678 | 369.76 | SCTF |
| El Hierro | 11147 | 278.71 | SCTF |
| Fuerteventura | 119732 | 1659 | LPGC |

La superficie (*Area*) está expresada en $km^2$ y las provincias corresponden con `LPGC`: Las Palmas de Gran Canaria y `SCTF`: Santa Cruz de Tenerife.

**Importante:** Nos referiremos a este DataFrame como `democan` de ahora en adelante.

## Gestión del índice

Cuando creamos un DataFrame, pandas autocompleta el índice con un valor entero autoincremental comenzando desde cero:

```
>>> pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
   A  B
0  1  3
1  2  4
```

Si queremos convertir alguna columna en el índice de la tabla, podemos hacerlo así:

```
>>> stats = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

>>> stats.set_index('A')  # columna A como índice
   B
```

(continué en la próxima página)

(proviene de la página anterior)

```
A
1   3
2   4
```

**Nota:** En el caso anterior se puede observar que el índice toma un nombre `A`. Esto se puede conseguir directamente asignando un valor a `df.index.name`.

Podemos añadir un parámetro (en la creación) para especificar los valores que queremos incluir en el índice:

```
>>> pd.DataFrame({'A': [1, 2], 'B': [3, 4]}, index=['R1', 'R2'])
    A  B
R1  1  3
R2  2  4
```

En aquellos DataFrames que disponen de un índice etiquetado, es posible resetearlo:

```
>>> pd.DataFrame({'A': [1, 2], 'B': [3, 4]}, index=['R1', 'R2']).reset_index()
  index  A  B
0    R1  1  3
1    R2  2  4
```

**Ejercicio**

Convierta la columna *Island* en el índice de `democan`. El DataFrame debería de quedar así:

```
>>> df
               Population     Area Province
Island
Gran Canaria       855521  1560.10     LPGC
Tenerife           928604  2034.38     SCTF
La Palma            83458   708.32     SCTF
Lanzarote          155812   845.94     LPGC
La Gomera           21678   369.76     SCTF
El Hierro           11147   278.71     SCTF
Fuerteventura      119732  1659.00     LPGC
```

### Lectura de fuentes externas

Lo más habitual cuando se trabaja en ciencia de datos es tener la información en distintas fuentes auxiliares: bases de datos, ficheros, llamadas remotas a APIs, etc. Pandas nos ofrece una variedad enorme de funciones para cargar datos desde, prácticamente, cualquier origen.

Tabla 2: Funciones para lectura de datos en pandas

| Función | Explicación |
|---|---|
| read_pickle | Lectura de datos en formato pickle (Python) |
| read_table | Lectura de ficheros con delimitadores |
| read_csv | Lectura de ficheros .csv |
| read_fwf | Lectura de tablas con líneas de ancho fijo |
| read_clipboard | Lectura de texto del portapapeles |
| read_excel | Lectura de ficheros excel |
| read_json | Lectura de ficheros json |
| read_html | Lectura de tablas HTML |
| read_xml | Lectura de documentos XML |
| read_hdf | Lectura de objetos pandas almacenados en fichero |
| read_feather | Lectura de objetos en formato «feather» |
| read_parquet | Lectura de objetos en formato «parquet» |
| read_orc | Lectura de objetos en formato ORC |
| read_sas | Lectura de ficheros SAS |
| read_spss | Lectura de ficheros SPSS |
| read_sql_table | Lectura de tabla SQL |
| read_sql_query | Lectura de una consulta SQL |
| read_sql | Wrapper para `read_sql_table` y `read_sql_query` |
| read_gbq | Lectura de datos desde Google BigQuery |
| read_stata | Lectura de ficheros Stata |
| | |
| | |

**Nota:** Todas estas funciones tienen su equivalente para escribir datos en los distintos formatos. En vez de `read_` habría que usar el prefijo `to_`. Por ejemplo: `.to_csv()`, `.to_json()` o `.to_sql()`

A modo de ilustración, vamos a leer el contenido del fichero `tech.csv` que contiene la lista de las mayores empresas tecnológicas por ingresos totales (en millones de dólares)[2].

Usaremos la función `read_csv()` que espera la **coma** como separador de campos. Este fichero está delimitado por tabuladores, por lo que especificaremos esta circunstancia mediante el parámetro `delimiter`. Igualmente, vamos a indicar que se use la primera columna *Company* como índice del DataFrame con el parámetro `index_col`:

```
>>> df = pd.read_csv('tech.csv', delimiter='\t', index_col='Company')

>>> df
                      Revenue  Employees              City        Country
Company
Apple                  274515     147000        California  United States
Samsung Electronics    200734     267937             Suwon    South Korea
Alphabet               182527     135301        California  United States
Foxconn                181945     878429    New Taipei City         Taiwan
Microsoft              143015     163000        Washington  United States
Huawei                 129184     197000          Shenzhen          China
Dell Technologies       92224     158000             Texas  United States
Facebook                85965      58604        California  United States
Sony                    84893     109700             Tokyo          Japan
Hitachi                 82345     350864             Tokyo          Japan
Intel                   77867     110600        California  United States
IBM                     73620     364800          New York  United States
Tencent                 69864      85858          Shenzhen          China
Panasonic               63191     243540             Osaka          Japan
Lenovo                  60742      71500         Hong Kong          China
HP Inc.                 56639      53000        California  United States
LG Electronics          53625      75000             Seoul    South Korea
```

**Truco:** Se suele usar `df` como nombre para las variables tipo DataFrame.

**Ejercicio**

Cargue el conjunto de datos `democan` desde `democan.csv` en un DataFrame `df` indicando que la columna *Island* es el índice.

*También es posible cargar el «dataset» a través de la URL que conseguimos con botón derecho: copiar enlace.*

## Características de un DataFrame

### Visualización de los datos

Para «echar un vistazo» a los datos, existen dos funciones muy recurridas:

```
>>> df.head()
                      Revenue  Employees              City        Country
```

(continué en la próxima página)

(proviene de la página anterior)

```
Company
Apple                  274515     147000       California  United States
Samsung Electronics    200734     267937            Suwon    South Korea
Alphabet               182527     135301       California  United States
Foxconn                181945     878429  New Taipei City         Taiwan
Microsoft              143015     163000       Washington  United States

>>> df.tail()
                Revenue  Employees        City        Country
Company
Tencent           69864      85858    Shenzhen          China
Panasonic         63191     243540       Osaka          Japan
Lenovo            60742      71500   Hong Kong          China
HP Inc.           56639      53000  California  United States
LG Electronics    53625      75000       Seoul    South Korea
```

**Truco:** Estas funciones admiten como parámetro el número de registros a visualizar.

## Información sobre los datos

Pandas ofrece algunas funciones que proporcionan un cierto «resumen» de los datos a nivel descriptivo. Veamos algunas de ellas.

Información sobre columnas:

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
Index: 17 entries, Apple to LG Electronics
Data columns (total 4 columns):
#   Column     Non-Null Count  Dtype
---  ------     --------------  -----
0   Revenue    17 non-null     int64
1   Employees  17 non-null     int64
2   City       17 non-null     object
3   Country    17 non-null     object
dtypes: int64(2), object(2)
memory usage: 680.0+ bytes
```

Descripción de las variables numéricas:

```
>>> df.describe()
             Revenue      Employees
```

(continué en la próxima página)

(proviene de la página anterior)

```
count      17.000000      17.000000
mean   112523.235294  204125.470588
std     63236.957691  198345.912495
min     53625.000000   53000.000000
25%     69864.000000   85858.000000
50%     84893.000000  147000.000000
75%    143015.000000  243540.000000
max    274515.000000  878429.000000
```

Uso de memoria:

```
>>> df.memory_usage()
Index        692
Revenue      136
Employees    136
City         136
Country      136
dtype: int64
```

**Truco:** El resultado de `describe()` es un DataFrame, mientras que el resultado de `memory_usage()` es Series. En cualquier caso, ambas estructuras son accesibles normalmente como tipos de datos Pandas.

### Atributos de un DataFrame

Tamaños y dimensiones:

```
>>> df.shape  # filas por columnas
(17, 4)

>>> df.size   # número total de datos
68

>>> df.ndim   # número de dimensiones
2
```

Índice, columnas y valores:

```
>>> df.index
Index(['Apple', 'Samsung Electronics', 'Alphabet', 'Foxconn', 'Microsoft',
       'Huawei', 'Dell Technologies', 'Facebook', 'Sony', 'Hitachi', 'Intel',
       'IBM', 'Tencent', 'Panasonic', 'Lenovo', 'HP Inc.', 'LG Electronics'],
```

(continué en la próxima página)

(proviene de la página anterior)

```
      dtype='object', name='Company')

>>> df.columns
Index(['Revenue', 'Employees', 'City', 'Country'], dtype='object')

>>> df.values
array([[274515, 147000, 'California', 'United States'],
       [200734, 267937, 'Suwon', 'South Korea'],
       [182527, 135301, 'California', 'United States'],
       [181945, 878429, 'New Taipei City', 'Taiwan'],
       [143015, 163000, 'Washington', 'United States'],
       [129184, 197000, 'Shenzhen', 'China'],
       [92224, 158000, 'Texas', 'United States'],
       [85965, 58604, 'California', 'United States'],
       [84893, 109700, 'Tokyo', 'Japan'],
       [82345, 350864, 'Tokyo', 'Japan'],
       [77867, 110600, 'California', 'United States'],
       [73620, 364800, 'New York', 'United States'],
       [69864, 85858, 'Shenzhen', 'China'],
       [63191, 243540, 'Osaka', 'Japan'],
       [60742, 71500, 'Hong Kong', 'China'],
       [56639, 53000, 'California', 'United States'],
       [53625, 75000, 'Seoul', 'South Korea']], dtype=object)
```

### Acceso a un DataFrame

Es fundamental conocer la estructura de un DataFrame para su adecuado manejo:

Para todos los ejemplos subsiguientes continuamos utilizando el conjunto de datos de empresas tecnológicas cargado previamente:

```
>>> df
                     Revenue  Employees             City        Country
Company
Apple                 274515     147000       California  United States
Samsung Electronics   200734     267937            Suwon    South Korea
Alphabet              182527     135301       California  United States
Foxconn               181945     878429  New Taipei City         Taiwan
Microsoft             143015     163000       Washington  United States
Huawei                129184     197000         Shenzhen          China
Dell Technologies      92224     158000            Texas  United States
Facebook               85965      58604       California  United States
Sony                   84893     109700            Tokyo          Japan
Hitachi                82345     350864            Tokyo          Japan
```

(continué en la próxima página)

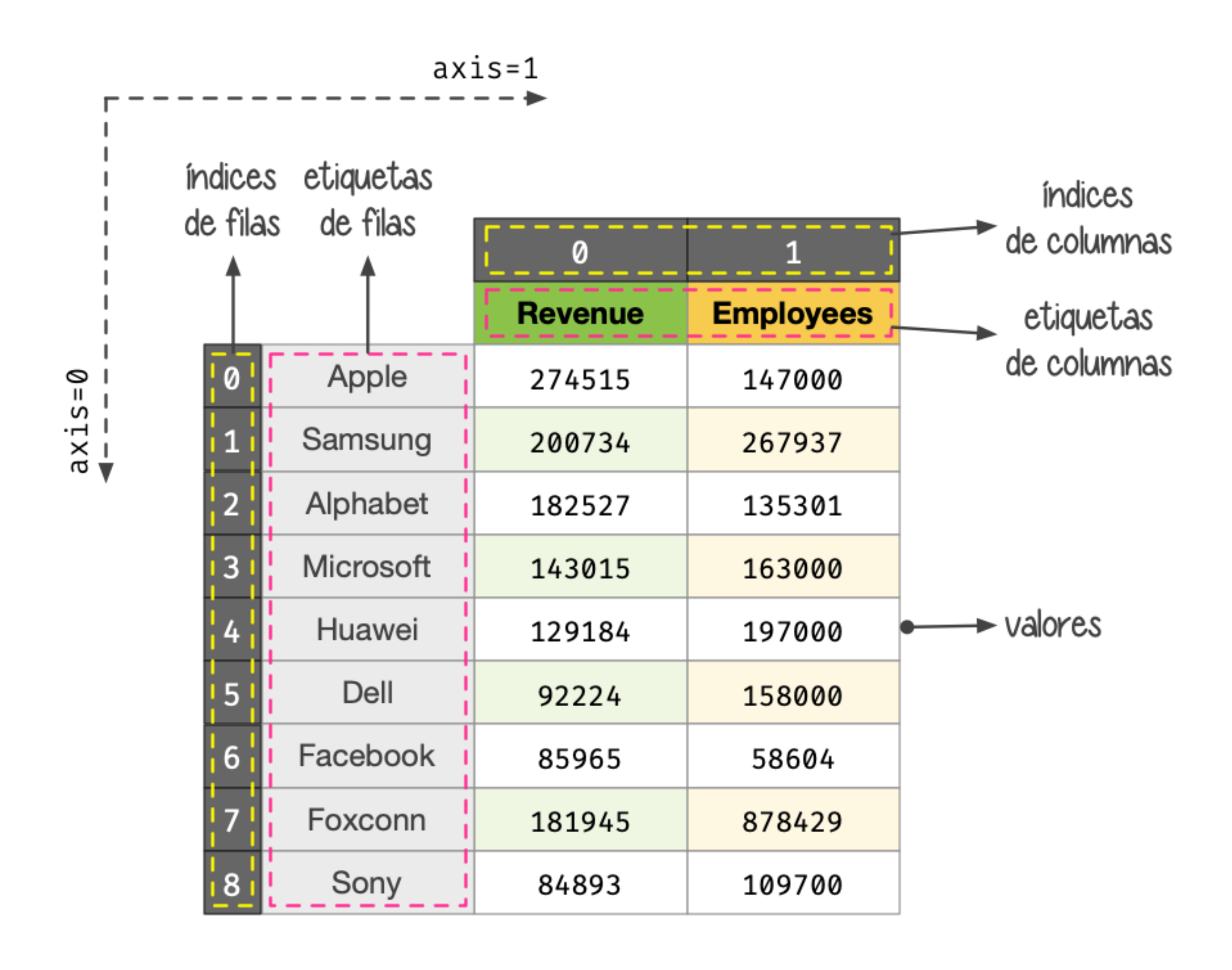



Figura 18: Componentes de un DataFrame

(proviene de la página anterior)

```
Intel                  77867     110600      California  United States
IBM                    73620     364800        New York  United States
Tencent                69864      85858        Shenzhen          China
Panasonic              63191     243540           Osaka          Japan
Lenovo                 60742      71500       Hong Kong          China
HP Inc.                56639      53000      California  United States
LG Electronics         53625      75000           Seoul    South Korea
```

### Acceso a filas

Si queremos acceder a las filas de un conjunto de datos **mediante la posición (índice numérico)** del registro usamos el atributo `iloc`:

```
>>> df.iloc[0]
Revenue              274515
Employees            147000
City             California
Country       United States
Name: Apple, dtype: object

>>> df.iloc[-1]
Revenue                53625
Employees              75000
City                   Seoul
Country          South Korea
Name: LG Electronics, dtype: object

>>> df.iloc[3:5]
         Revenue  Employees             City        Country
Company
Foxconn     181945     878429  New Taipei City         Taiwan
Microsoft   143015     163000       Washington  United States

>>> df.iloc[::5]  # Salto de 5 en 5 filas
         Revenue  Employees        City        Country
Company
Apple     274515     147000  California  United States
Huawei    129184     197000    Shenzhen          China
Intel      77867     110600  California  United States
HP Inc.    56639      53000  California  United States
```

**Nota:** El acceso a un registro individual nos devuelve una serie.

Si queremos acceder a las filas de un conjunto de datos **mediante la etiqueta del registro** usamos el atributo `loc`:

```
>>> df.loc['Apple']
Revenue               274515
Employees             147000
City              California
Country        United States
Name: Apple, dtype: object

>>> df.loc['IBM']
Revenue                73620
Employees             364800
City                New York
Country        United States
Name: IBM, dtype: object

>>> df.loc['Sony':'Intel']
         Revenue  Employees        City        Country
Company
Sony       84893     109700       Tokyo          Japan
Hitachi    82345     350864       Tokyo          Japan
Intel      77867     110600  California  United States
```

**Nota:** El acceso a un registro individual nos devuelve una serie.

### Acceso a columnas

El acceso a columnas se realiza directamente utilizando corchetes, como si fuera un diccionario:

```
>>> df['Revenue']  # equivalente a df.Revenue
Company
Apple                   274515
Samsung Electronics     200734
Alphabet                182527
Foxconn                 181945
Microsoft               143015
Huawei                  129184
Dell Technologies        92224
Facebook                 85965
Sony                     84893
Hitachi                  82345
```

(continué en la próxima página)

(proviene de la página anterior)

```
Intel                   77867
IBM                     73620
Tencent                 69864
Panasonic               63191
Lenovo                  60742
HP Inc.                 56639
LG Electronics          53625
Name: Revenue, dtype: int64
```

**Nota:** El acceso a una columna individual nos devuelve una serie.

Se pueden seleccionar varias columnas a la vez pasando una lista:

```
>>> df[['Employees', 'City']].head()
                     Employees            City
Company
Apple                   147000      California
Samsung Electronics     267937           Suwon
Alphabet                135301      California
Foxconn                 878429  New Taipei City
Microsoft               163000      Washington
```

Esta misma sintaxis permite la **reordenación de las columnas** de un DataFrame, si asignamos el resultado a la misma (u otra) variable:

```
>>> df_reordered = df[['City', 'Country', 'Revenue', 'Employees']]

>>> df_reordered.head()
                                City        Country  Revenue  Employees
Company
Apple                     California  United States   274515     147000
Samsung Electronics            Suwon    South Korea   200734     267937
Alphabet                  California  United States   182527     135301
Foxconn              New Taipei City         Taiwan   181945     878429
Microsoft                 Washington  United States   143015     163000
```

### Acceso a filas y columnas

Si mezclamos los dos accesos anteriores podemos seleccionar datos de forma muy precisa. Como siempre, partimos del «dataset» de empresas tecnológicas:

```
>>> df.head()
                     Revenue  Employees           City        Country
Company
Apple                 274515     147000     California  United States
Samsung Electronics   200734     267937          Suwon    South Korea
Alphabet              182527     135301     California  United States
Foxconn               181945     878429  New Taipei City         Taiwan
Microsoft             143015     163000     Washington  United States
```

Acceso al **primer valor del número de empleados/as**. Formas equivalentes de hacerlo:

```
>>> df.iloc[0, 0]
274515

>>> df.loc['Apple', 'Revenue']
274515
```

Acceso a **ciudad y país de las empresas Sony, Panasonic y Lenovo**:

```
>>> df.loc[['Sony', 'Panasonic', 'Lenovo'], ['City', 'Country']]
                City Country
Company
Sony           Tokyo   Japan
Panasonic      Osaka   Japan
Lenovo     Hong Kong   China
```

Acceso a la **última columna** del DataFrame:

```
>>> df.iloc[:, -1]
Company
Apple                  United States
Samsung Electronics      South Korea
Alphabet               United States
Foxconn                       Taiwan
Microsoft              United States
Huawei                         China
Dell Technologies      United States
Facebook               United States
Sony                           Japan
Hitachi                        Japan
Intel                  United States
```

(continué en la próxima página)

(proviene de la página anterior)

```
IBM                    United States
Tencent                        China
Panasonic                      Japan
Lenovo                         China
HP Inc.                United States
LG Electronics           South Korea
Name: Country, dtype: object
```

Acceso a las **tres últimas filas (empresas) y a las dos primeras columnas**:

```
>>> df.iloc[-3:, :2]
                Revenue  Employees
Company
Lenovo            60742      71500
HP Inc.           56639      53000
LG Electronics    53625      75000
```

Acceso a **las filas que van desde «Apple» a «Huawei» y a las columnas que van desde «Revenue» hasta «City»**:

```
>>> df.loc['Apple':'Huawei', 'Revenue':'City']
                     Revenue  Employees             City
Company
Apple                 274515     147000       California
Samsung Electronics   200734     267937            Suwon
Alphabet              182527     135301       California
Foxconn               181945     878429  New Taipei City
Microsoft             143015     163000       Washington
Huawei                129184     197000         Shenzhen
```

**Truco:** Es posible usar «slicing» (troceado) en el acceso a registros y columnas.

### Selección condicional

Es posible aplicar ciertas condiciones en la selección de los datos para obtener el subconjunto que estemos buscando. Veremos distintas aproximaciones a esta técnica.

Supongamos que queremos seleccionar aquellas **empresas con base en Estados Unidos**. Si aplicamos la condición sobre la columna obtendremos una serie de tipo «booleano» en la que se indica para qué registros se cumple la condición (incluyendo el índice):

```
>>> df['Country'] == 'United States'
Company
Apple                    True
Samsung Electronics     False
Alphabet                 True
Foxconn                 False
Microsoft                True
Huawei                  False
Dell Technologies        True
Facebook                 True
Sony                    False
Hitachi                 False
Intel                    True
IBM                      True
Tencent                 False
Panasonic               False
Lenovo                  False
HP Inc.                  True
LG Electronics          False
Name: Country, dtype: bool
```

Si aplicamos esta «máscara» al conjunto original de datos, obtendremos las empresas que estamos buscando:

```
>>> df[df['Country'] == 'United States']
                   Revenue  Employees        City        Country
Company
Apple               274515     147000  California  United States
Alphabet            182527     135301  California  United States
Microsoft           143015     163000  Washington  United States
Dell Technologies    92224     158000       Texas  United States
Facebook             85965      58604  California  United States
Intel                77867     110600  California  United States
IBM                  73620     364800    New York  United States
HP Inc.              56639      53000  California  United States
```

También es posible aplicar condiciones compuestas. Supongamos que necesitamos selecionar aquellas **empresas con más de 100000 millones de dólares de ingresos y más de 100000 empleados/as**:

```
>>> revenue_condition = df['Revenue'] > 100_000
>>> employees_condition = df['Employees'] > 100_000

>>> df[revenue_condition & employees_condition]
                     Revenue  Employees             City        Country
Company
```

(continué en la próxima página)

(proviene de la página anterior)

```
Apple                 274515     147000       California  United States
Samsung Electronics   200734     267937            Suwon    South Korea
Alphabet              182527     135301       California  United States
Foxconn               181945     878429  New Taipei City         Taiwan
Microsoft             143015     163000       Washington  United States
Huawei                129184     197000         Shenzhen          China
```

Los operadores lógicos que se pueden utilizar para combinar condiciones de selección son los siguientes:

| Operador | Significado |
|---|---|
| \| | «or» lógico |
| & | «and» lógico |
| ~ | «not» lógico |
| ^ | «xor» lógico |

Imaginemos ahora que estamos buscando aquellas **empresas establecidas en California o Tokyo**. Una posible aproximación sería utilizar una condición compuesta, pero existe la función `isin()` que nos permite comprobar si un valor está dentro de una lista de opciones:

```
>>> mask = df['City'].isin(['California', 'Tokyo'])

>>> df[mask]
          Revenue  Employees        City        Country
Company
Apple      274515     147000  California  United States
Alphabet   182527     135301  California  United States
Facebook    85965      58604  California  United States
Sony        84893     109700       Tokyo          Japan
Hitachi     82345     350864       Tokyo          Japan
Intel       77867     110600  California  United States
HP Inc.     56639      53000  California  United States
```

---

### Ejercicio

Obtenga los siguientes subconjuntos del «dataset» `democan`:

```
# Use .loc
           Population    Area Province
Island
El Hierro       11147  278.71     SCTF
La Gomera       21678  369.76     SCTF
```

```
# Use .loc
Island
Gran Canaria     LPGC
Tenerife         SCTF
La Palma         SCTF
Lanzarote        LPGC
La Gomera        SCTF
El Hierro        SCTF
Fuerteventura    LPGC
Name: Province, dtype: object
```

```
# Use .iloc
Island
Gran Canaria     1560.10
La Palma          708.32
La Gomera         369.76
Fuerteventura    1659.00
Name: Area, dtype: float64
```

```
# Islas con más de 1000 km2 de extensión
               Population     Area Province
Island
Gran Canaria       855521  1560.10     LPGC
Tenerife           928604  2034.38     SCTF
Fuerteventura      119732  1659.00     LPGC
```

## Seleción usando «query»

Pandas provee una alternativa para la selección condicional de registros a través de la función query(). Admite una sintaxis de consulta mediante operadores de comparación.

Veamos las mismas consultas de ejemplo que para el apartado anterior:

```
>>> df.query('Country == "United States"')
                Revenue  Employees        City        Country
Company
Apple                274515     147000  California  United States
Alphabet             182527     135301  California  United States
Microsoft            143015     163000  Washington  United States
Dell Technologies     92224     158000       Texas  United States
Facebook              85965      58604  California  United States
Intel                 77867     110600  California  United States
IBM                   73620     364800    New York  United States
```

(continué en la próxima página)

(proviene de la página anterior)

```
HP Inc.                 56639      53000  California  United States

>>> df.query('Revenue > 100_000 & Employees > 100_000')
                     Revenue  Employees             City        Country
Company
Apple                 274515     147000       California  United States
Samsung Electronics   200734     267937            Suwon    South Korea
Alphabet              182527     135301       California  United States
Foxconn               181945     878429  New Taipei City         Taiwan
Microsoft             143015     163000       Washington  United States
Huawei                129184     197000         Shenzhen          China

>>> df.query('City in ["California", "Tokyo"]')
          Revenue  Employees        City        Country
Company
Apple      274515     147000  California  United States
Alphabet   182527     135301  California  United States
Facebook    85965      58604  California  United States
Sony        84893     109700       Tokyo          Japan
Hitachi     82345     350864       Tokyo          Japan
Intel       77867     110600  California  United States
HP Inc.     56639      53000  California  United States
```

**Truco:** Si los nombres de columna contienen espacios, se puede hacer referencias a ellas con comillas invertidas. Por ejemplo: `` `Total Stock` ``.

### Comparativa en consultas

Hemos visto dos métodos para realizar consultas (o filtrado) en un DataFrame: usando selección booleana con corchetes y usando la función `query`. ¿Ambos métodos son igual de eficientes en términos de rendimiento?

Haremos una comparativa muy simple para tener, al menos, una idea de sus órdenes de magnitud. En primer lugar creamos un DataFrame con 3 columnas y 1 millón de valores aleatorios enteros en cada una de ellas:

```
>>> size = 1_000_000

>>> data = {
...     'A': np.random.randint(1, 100, size=size),
...     'B': np.random.randint(1, 100, size=size),
...     'C': np.random.randint(1, 100, size=size)
```

(continué en la próxima página)

(proviene de la página anterior)

```
... }

>>> df = pd.DataFrame(data)

>>> df.shape
(1000000, 3)
```

Ahora realizaremos la misma consulta sobre el DataFrame aplicando los métodos ya vistos:

```
>>> %timeit df[(df['A'] > 50) & (df['B'] < 50)]
5.86 ms ± 28.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

>>> %timeit df.query('A > 50 & B < 50')
7.54 ms ± 115 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Sin que esto sea en modo alguna concluyente, da la sensación de que `query()` añade un cierto «overhead»[7] al filtrado y aumentan los tiempos de cómputo.

## Modificación de un DataFrame

### Modificando valores existentes

Partiendo del *acceso a los datos* que ya hemos visto, podemos asignar valores sin mayor dificultad.

Pero antes de modificar el DataFrame original, vamos a hacer una copia del mismo:

```
>>> df_mod = df.copy()

>>> df_mod.equals(df)  # comprueba que todos los valores del DataFrame son iguales
True
```

Supongamos que hemos cometido un **error en el número de empleados/as de Apple** y queremos corregirlo:

```
>>> df_mod.head(1)
         Revenue  Employees        City        Country
Company
Apple     274515     147000  California  United States

>>> df_mod.loc['Apple', 'Employees'] = 137000
```

(continué en la próxima página)

[7] Exceso de tiempo de cómputacion, memoria o ancho de banda que son necesarios para realizar una tarea específica.

(proviene de la página anterior)

```
>>> df_mod.head(1)
        Revenue  Employees        City        Country
Company
Apple    274616     137000  California  United States
```

Supongamos que no se había contemplado una **subida del 20% en los ingresos** y queremos reflejarla:

```
>>> df_mod['Revenue'] *= 1.20

>>> df_mod['Revenue'].head()
Company
Apple                  329418.0
Samsung Electronics    240880.8
Alphabet               219032.4
Foxconn                218334.0
Microsoft              171618.0
Name: Revenue, dtype: float64
```

Supongamos que todas las empresas tecnológicas **mueven su sede a Vigo (España)** y queremos reflejarlo:

```
>>> df_mod['City'] = 'Vigo'
>>> df_mod['Country'] = 'Spain'

>>> df_mod.head()
                      Revenue  Employees  City Country
Company
Apple                329418.0     137000  Vigo   Spain
Samsung Electronics  240880.8     267937  Vigo   Spain
Alphabet             219032.4     135301  Vigo   Spain
Foxconn              218334.0     878429  Vigo   Spain
Microsoft            171618.0     163000  Vigo   Spain
```

**Nota:** En este último ejemplo se produce un «broadcast» o difusión del valor escalar a todos los registros del «dataset».

### Reemplazo de valores

Hay una función muy importante en lo relativo a la modificación de valores. Se trata de `replace()` y admite una amplia variedad de parámetros. Se puede usar tanto para tipos numéricos como textuales.

Uno de los usos más habituales es la recodificación. Supongamos que queremos **recodificar los países en ISO3166 Alpha-3** para el DataFrame de empresas tecnológicas:

```
>>> iso3166 = {
    'United States': 'USA',
    'South Korea': 'KOR',
    'Taiwan': 'TWN',
    'China': 'CHN',
    'Japan': 'JPN'
}

>>> df['Country'].replace(iso3166)
Company
Apple                  USA
Samsung Electronics    KOR
Alphabet               USA
Foxconn                TWN
Microsoft              USA
Huawei                 CHN
Dell Technologies      USA
Facebook               USA
Sony                   JPN
Hitachi                JPN
Intel                  USA
IBM                    USA
Tencent                CHN
Panasonic              JPN
Lenovo                 CHN
HP Inc.                USA
LG Electronics         KOR
Name: Country, dtype: object
```

**Ejercicio**

Recodifique la columna *Province* del «dataset» `democan` de tal manera que aparezcan las provincias con el texto completo: *Santa Cruz de Tenerife* y *Las Palmas de Gran Canaria*.

### Insertando y borrando filas

Podemos insertar datos en un DataFrame como filas o como columnas.

Supongamos que queremos incluir una **nueva empresa Cisco**[4]:

```
>>> cisco = pd.Series(data=[51_904, 75_900, 'California', 'United States'],
...                   index=df_mod.columns, name='Cisco')

>>> cisco
Revenue                51904
Employees              75900
City              California
Country        United States
Name: Cisco, dtype: object

>>> df_mod = df_mod.append(cisco)

>>> df_mod.tail(3)
                Revenue  Employees        City        Country
Company
HP Inc.         67966.8      53000        Vigo          Spain
LG Electronics  64350.0      75000        Vigo          Spain
Cisco           51904.0      75900  California  United States
```

**Truco:** El método `append()` devuelve un nuevo DataFrame con los datos añadidos. Es por eso que si queremos consolidar los cambios, debemos realizar una asignación.

Imaginemos ahora que **Facebook, Tencent e Hitachi caen en bancarrota** y debemos eliminarlas de nuestro conjunto de datos:

```
>>> df_mod = df_mod.drop(labels=['Facebook', 'Tencent', 'Hitachi'])

>>> df_mod.index  # ya no aparecen en el índice
Index(['Apple', 'Samsung Electronics', 'Alphabet', 'Foxconn', 'Microsoft',
    'Huawei', 'Dell Technologies', 'Sony', 'Intel', 'IBM', 'Panasonic',
    'Lenovo', 'HP Inc.', 'LG Electronics', 'Cisco'],
    dtype='object', name='Company')
```

[4] Datos del año 2020 según Wikipedia.

### Insertando y borrando columnas

Insertar una columna en un DataFrame es equivalente a *añadir una clave en un diccionario*.

Supongamos que queremos **añadir una columna «Expenses» (gastos)**. No manejamos esta información, así que, a modo de ejemplo, utilizaremos unos valores aleatorios:

```
>>> expenses = np.random.randint(50_000, 300_000, size=15)

>>> expenses
array([139655,  97509, 220777, 260609, 121145, 112338,  72815, 159843,
    205695,  97672,  89614, 260028, 171650, 152049,  57006])

>>> df_mod['Expenses'] = expenses

>>> df_mod.head()
                      Revenue  Employees  City Country  Expenses
Company
Apple                329418.0     137000  Vigo   Spain    139655
Samsung Electronics  240880.8     267937  Vigo   Spain     97509
Alphabet             219032.4     135301  Vigo   Spain    220777
Foxconn              218334.0     878429  Vigo   Spain    260609
Microsoft            171618.0     163000  Vigo   Spain    121145
```

**Truco:** También existe la función insert() que nos permite insertar una columna en una posición determinada.

En el caso de que no nos haga falta una columna podemos borrarla fácilmente. Una opción sería utilizar la sentencia `del`, pero seguiremos con el uso de funciones propias de pandas. Imaginemos que queremos **eliminar la columna «Expenses»**:

```
>>> df_mod.columns
Index(['Revenue', 'Employees', 'City', 'Country', 'Expenses'], dtype='object')

>>> df_mod = df_mod.drop(labels='Expenses', axis=1)

>>> df_mod.columns
Index(['Revenue', 'Employees', 'City', 'Country'], dtype='object')
```

**Truco:** Recordar que el parámetro `axis` indica en qué «dirección» estamos trabajando. Véase *el acceso a un DataFrame*.

### El parámetro inplace

Muchas de las funciones de pandas se dicen «no destructivas» en el sentido de que no modifican el conjunto de datos original, sino que devuelven uno nuevo con las modificaciones realizadas. Pero este comportamiento se puede modificar utilizando el parámetro `inplace`.

Veamos un ejemplo con el borrado de columnas:

```
>>> df_mod.head()
                      Revenue  Employees  City Country
Company
Apple                329418.0     137000  Vigo   Spain
Samsung Electronics  240880.8     267937  Vigo   Spain
Alphabet             219032.4     135301  Vigo   Spain
Foxconn              218334.0     878429  Vigo   Spain
Microsoft            171618.0     163000  Vigo   Spain

>>> df_mod.drop(labels=['City', 'Country'], axis=1, inplace=True)

>>> df_mod.head()
                      Revenue  Employees
Company
Apple                329418.0     137000
Samsung Electronics  240880.8     267937
Alphabet             219032.4     135301
Foxconn              218334.0     878429
Microsoft            171618.0     163000
```

**Ejercicio**

Añada una nueva columna *Density* a `democan` de tal manera que represente la densidad de población de cada isla del archipiélago canario.

También es posible **renombrar columnas** utilizando la función rename() de Pandas.

Supongamos un caso de uso en el que queremos **renombrar las columnas a sus tres primeras letras en minúsculas**. Tenemos dos maneras de hacerlo. La primera sería directamente creando un «mapping» entre los nombres de columna actuales y los nombres nuevos:

```
>>> new_columns = {'Revenue': 'rev', 'Employees': 'emp', 'City': 'cit', 'Country':
↪'cou'}

>>> df.rename(columns=new_columns).head(3)
                        rev     emp          cit             cou
```

(continué en la próxima página)

(proviene de la página anterior)

```
Company
Apple                 274515  147000  California  United States
Samsung Electronics   200734  267937       Suwon    South Korea
Alphabet              182527  135301  California  United States
```

Otro camino para conseguir el mismo resultado es aplicar una función que realice esta tarea de manera automatizada:

```
>>> df.rename(columns=lambda c: c.lower()[:3]).head(3)
                        rev     emp         cit            cou
Company
Apple                 274515  147000  California  United States
Samsung Electronics   200734  267937       Suwon    South Korea
Alphabet              182527  135301  California  United States
```

**Ver también:**

Si en vez del parámetro nominal `columns` utilizamos el parámetro `index` estaremos renombrando los valores del índice. Se aplica el mismo comportamiento ya visto.

Nada impide **asignar directamente una lista (tupla) de nombres a las columnas** de un DataFrame:

```
>>> df.columns = ('Ingresos', 'Empleados', 'Ciudad', 'País')

>>> df.head(3)
                     Ingresos  Empleados      Ciudad           País
Company
Apple                  274515     147000  California  United States
Samsung Electronics    200734     267937       Suwon    South Korea
Alphabet               182527     135301  California  United States
```

## Otras operaciones con un DataFrame

### Manejando cadenas de texto

A menudo solemos trabajar con datos que incluyen información textual. Pandas también nos ofrece herramientas para cubrir estos casos.

De hecho, simplemente debemos utilizar el manejador `str` y tendremos a disposición la gran mayoría de funciones vistas en la sección de *cadenas de texto*.

Veamos un primer ejemplo en el que **pasamos a mayúsculas las ciudades en las que se localizan las empresas** tecnológicas:

```
>>> df['City'].str.upper()
Company
Apple                      CALIFORNIA
Samsung Electronics             SUWON
Alphabet                   CALIFORNIA
Foxconn               NEW TAIPEI CITY
Microsoft                  WASHINGTON
Huawei                       SHENZHEN
Dell Technologies               TEXAS
Facebook                   CALIFORNIA
Sony                            TOKYO
Hitachi                         TOKYO
Intel                      CALIFORNIA
IBM                          NEW YORK
Tencent                      SHENZHEN
Panasonic                       OSAKA
Lenovo                      HONG KONG
HP Inc.                    CALIFORNIA
LG Electronics                  SEOUL
Name: City, dtype: object
```

Otro supuesto sería el de **sustituir espacios por subguiones en los países de las empresas**:

```
>>> df['Country'].str.replace(' ', '_')
Company
Apple                  United_States
Samsung Electronics      South_Korea
Alphabet               United_States
Foxconn                       Taiwan
Microsoft              United_States
Huawei                         China
Dell Technologies      United_States
Facebook               United_States
Sony                           Japan
Hitachi                        Japan
Intel                  United_States
IBM                    United_States
Tencent                        China
Panasonic                      Japan
Lenovo                         China
HP Inc.                United_States
LG Electronics           South_Korea
Name: Country, dtype: object
```

### Expresiones regulares

El uso de expresiones regulares aporta una gran expresividad. Veamos su aplicación con tres casos de uso:

- Filtrado de filas.
- Reemplazo de valores.
- Extracción de columnas.

Supongamos que queremos **filtrar las empresas y quedarnos con las que comienzan por vocal**:

```
>>> mask = df.index.str.match(r'^[aeiou]', flags=re.IGNORECASE)

>>> df[mask]
          Revenue  Employees        City        Country
Company
Apple      274515     147000  California  United States
Alphabet   182527     135301  California  United States
Intel       77867     110600  California  United States
IBM         73620     364800    New York  United States
```

---

**Nota:** Dado que el nombre de la empresa está actuando como índice del «dataset», hemos aplicado la búsqueda sobre `.index`.

---

Ahora imaginemos que vamos a **sustituir aquellas ciudades que empiezan con «S» o «T» por «Stanton»**:

```
>>> df['City'].str.replace(r'^[ST].*', 'Stanton', regex=True)
Company
Apple                   California
Samsung Electronics        Stanton
Alphabet                California
Foxconn                New Stanton
Microsoft               Washington
Huawei                     Stanton
Dell Technologies          Stanton
Facebook                California
Sony                       Stanton
Hitachi                    Stanton
Intel                   California
IBM                       New York
Tencent                    Stanton
Panasonic                    Osaka
```

(continué en la próxima página)

(proviene de la página anterior)

```
Lenovo                       Hong Kong
HP Inc.                     California
LG Electronics                 Stanton
Name: City, dtype: object
```

Por último supongamos que queremos **dividir la columna «Country»** en dos columnas usando el espacio como separador:

```
>>> df['Country'].str.split(' ', expand=True)
                          0       1
Company
Apple                United  States
Samsung Electronics   South   Korea
Alphabet             United  States
Foxconn              Taiwan    None
Microsoft            United  States
Huawei                China    None
Dell Technologies    United  States
Facebook             United  States
Sony                  Japan    None
Hitachi               Japan    None
Intel                United  States
IBM                  United  States
Tencent               China    None
Panasonic             Japan    None
Lenovo                China    None
HP Inc.              United  States
LG Electronics        South   Korea
```

Existen otras funciones interesantes de Pandas que trabajan sobre expresiones regulares:

- count() para contar el número de ocurrencias de un patrón.
- contains() para comprobar si existe un determinado patrón.
- extract() para extraer grupos de captura sobre un patrón.
- findall() para encontrar todas las ocurrencias de un patrón.

### Manejando fechas

Suele ser habitual tener que manejar datos en formato fecha (o fecha-hora). Pandas ofrece un amplio abanico de posibilidades para ello. Veamos algunas de las herramientas disponibles.

Para ejemplificar este apartado hemos añadido al «dataset» de empresas tecnológicas una nueva columna con las fechas de fundación de las empresas (en formato «string»):

```
>>> df['Founded'] = ['1/4/1976',   '13/1/1969', '4/9/1998',  '20/2/1974',
...                  '4/4/1975',   '15/9/1987', '1/2/1984',  '4/2/2004',
...                  '7/5/1946',   '1/10/1962', '18/7/1968', '16/6/1911',
...                  '11/11/1998', '13/3/1918', '1/11/1984', '1/1/1939',
...                  '5/1/1947']

>>> df.head()
                    Revenue  Employees            City        Country    Founded
Company
Apple                274515     147000      California  United States   1/4/1976
Samsung Electronics  200734     267937           Suwon    South Korea  13/1/1969
Alphabet             182527     135301      California  United States   4/9/1998
Foxconn              181945     878429  New Taipei City         Taiwan  20/2/1974
Microsoft            143015     163000      Washington  United States   4/4/1975

>>> df['Founded'].dtype  # tipo "object"
dtype('O')
```

Lo primero que deberíamos hacer es convertir la columna «Founded» al tipo «datetime» usando la función to_datetime():

```
>>> df['Founded'] = pd.to_datetime(df['Founded'])

>>> df['Founded'].head()
Company
Apple                  1976-01-04
Samsung Electronics    1969-01-13
Alphabet               1998-04-09
Foxconn                1974-02-20
Microsoft              1975-04-04
Name: Founded, dtype: datetime64[ns]
```

Es posible acceder a cada elemento de la fecha:

```
>>> df['fyear'] = df['Founded'].dt.year
>>> df['fmonth'] = df['Founded'].dt.month
>>> df['fday'] = df['Founded'].dt.day

>>> df.loc[:, 'Founded':].head()
```

(continué en la próxima página)

(proviene de la página anterior)

```
                       Founded  fyear  fmonth  fday
Company
Apple               1976-01-04   1976       1     4
Samsung Electronics 1969-01-13   1969       1    13
Alphabet            1998-04-09   1998       4     9
Foxconn             1974-02-20   1974       2    20
Microsoft           1975-04-04   1975       4     4
```

Por ejemplo, podríamos querer calcular el **número de años que llevan activas las empresas**:

```
>>> pd.to_datetime('today').year - df['Founded'].dt.year
Company
Apple                   46
Samsung Electronics     53
Alphabet                24
Foxconn                 48
Microsoft               47
Huawei                  35
Dell Technologies       38
Facebook                18
Sony                    76
Hitachi                 60
Intel                   54
IBM                    111
Tencent                 24
Panasonic              104
Lenovo                  38
HP Inc.                 83
LG Electronics          75
Name: Founded, dtype: int64
```

Los tipos de datos «datetime» dan mucha flexibilidad a la hora de hacer consultas:

```
>>> # Empresas creadas antes de 1950
>>> df.query('Founded <= 1950')
                Revenue  Employees        City        Country     Founded
Company
Sony              84893     109700       Tokyo          Japan 1946-07-05
IBM               73620     364800    New York  United States 1911-06-16
Panasonic         63191     243540       Osaka          Japan 1918-03-13
HP Inc.           56639      53000  California  United States 1939-01-01
LG Electronics    53625      75000       Seoul    South Korea 1947-05-01

>>> # Empresas creadas en Enero
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> df.query('Founded.dt.month == 1')
                     Revenue  Employees        City        Country    Founded
Company
Apple                 274515     147000  California  United States 1976-01-04
Samsung Electronics   200734     267937       Suwon    South Korea 1969-01-13
Dell Technologies      92224     158000       Texas  United States 1984-01-02
Hitachi                82345     350864       Tokyo          Japan 1962-01-10
Lenovo                 60742      71500   Hong Kong          China 1984-01-11
HP Inc.                56639      53000  California  United States 1939-01-01

>>> # Empresas creadas en el último cuatrimestre del año
>>> df.query('9 <= Founded.dt.month <= 12')
         Revenue  Employees      City Country    Founded
Company
Huawei    129184     197000  Shenzhen   China 1987-09-15
Tencent    69864      85858  Shenzhen   China 1998-11-11
```

Hay ocasiones en las que necesitamos que la fecha se convierta en el índice del DataFrame:

```
>>> df = df.reset_index().set_index('Founded').sort_index()

>>> df.head()
                   Company  Revenue  Employees        City        Country
Founded
1911-06-16             IBM    73620     364800    New York  United States
1918-03-13       Panasonic    63191     243540       Osaka          Japan
1939-01-01         HP Inc.    56639      53000  California  United States
1946-07-05            Sony    84893     109700       Tokyo          Japan
1947-05-01  LG Electronics    53625      75000       Seoul    South Korea
```

Esto nos permite indexar de forma mucho más precisa:

```
>>> # Empresas creadas en 1988
>>> df.loc['1998']
             Company  Revenue  Employees        City        Country
Founded
1998-04-09  Alphabet   182527     135301  California  United States
1998-11-11   Tencent    69864      85858    Shenzhen          China

>>> # Empresas creadas entre 1970 y 1980
>>> df.loc['1970':'1980']
              Company  Revenue  Employees             City        Country
Founded
1974-02-20    Foxconn   181945     878429  New Taipei City         Taiwan
1975-04-04  Microsoft   143015     163000       Washington  United States
```

(continué en la próxima página)

(proviene de la página anterior)

```
1976-01-04      Apple   274515     147000       California  United States

>>> # Empresas creadas entre enero de 1975 y marzo de 1984
>>> df.loc['1975-1':'1984-3']
                      Company  Revenue  Employees        City        Country
Founded
1975-04-04          Microsoft   143015     163000  Washington  United States
1976-01-04              Apple   274515     147000  California  United States
1984-01-02  Dell Technologies    92224     158000       Texas  United States
1984-01-11             Lenovo    60742      71500   Hong Kong          China
```

**Ejercicio**

Partiendo del fichero `oasis.csv` que contiene información sobre la discografía del grupo de pop británico Oasis, se pide:

- Cargue el fichero en un DataFrame.
- Convierta la columna «album_release_date» a tipo «datetime».
- Obtenga los nombres de los álbumes publicados entre 2000 y 2005.

### Manejando categorías

Hasta ahora hemos visto tipos de datos numéricos, cadenas de texto y fechas. ¿Pero qué ocurre con las categorías?

Las categorías pueden ser tanto datos numéricos como textuales, con la característica de tener un número discreto (relativamente pequeño) de elementos y, en ciertas ocasiones, un orden preestablecido. Ejemplos de variables categóricas son: género, idioma, meses del año, color de ojos, nivel de estudios, grupo sanguíneo, valoración, etc.

Pandas facilita el tratamiento de datos categóricos mediante un tipo específico `Categorical`.

Siguiendo con el «dataset» de empresas tecnológicas, vamos a añadir el continente al que pertenece cada empresa. En primera instancia mediante valores de texto habituales:

```
>>> df['Continent'] = ['America', 'Asia', 'America', 'Asia',
...                    'America', 'Asia', 'America', 'America',
...                    'Asia',    'Asia', 'America', 'America',
...                    'Asia',    'Asia', 'Asia',    'America',
...                    'Asia']

>>> df['Continent'].head()
```

(continué en la próxima página)

(proviene de la página anterior)

```
Company
Apple                     America
Samsung Electronics          Asia
Alphabet                  America
Foxconn                      Asia
Microsoft                 America
Name: Continent, dtype: object
```

Ahora podemos convertir esta columna a tipo categoría:

```
>>> df['Continent'].astype('category')
Company
Apple                     America
Samsung Electronics          Asia
Alphabet                  America
Foxconn                      Asia
Microsoft                 America
Huawei                       Asia
Dell Technologies         America
Facebook                  America
Sony                         Asia
Hitachi                      Asia
Intel                     America
IBM                       America
Tencent                      Asia
Panasonic                    Asia
Lenovo                       Asia
HP Inc.                   America
LG Electronics               Asia
Name: Continent, dtype: category
Categories (2, object): ['America', 'Asia']
```

En este caso, al ser una conversión «automática», las categorías no han incluido ningún tipo de orden. Pero imaginemos que queremos establecer un orden para las categorías de continentes basadas, por ejemplo, en su población: Asia, África, Europa, América, Australia:

```
>>> from pandas.api.types import CategoricalDtype

>>> continents = ('Asia', 'Africa', 'Europe', 'America', 'Australia')

>>> cat_continents = CategoricalDtype(categories=continents, ordered=True)

>>> df['Continent'].astype(cat_continents)
Company
Apple                     America
```

(continué en la próxima página)

(proviene de la página anterior)

```
Samsung Electronics       Asia
Alphabet               America
Foxconn                   Asia
Microsoft              America
Huawei                    Asia
Dell Technologies      America
Facebook               America
Sony                      Asia
Hitachi                   Asia
Intel                  America
IBM                    America
Tencent                   Asia
Panasonic                 Asia
Lenovo                    Asia
HP Inc.                America
LG Electronics            Asia
Name: Continent, dtype: category
Categories (5, object): ['Asia' < 'Africa' < 'Europe' < 'America' < 'Australia']
```

El hecho de trabajar con **categorías ordenadas** permite (entre otras) estas operaciones:

```
>>> df['Continent'].min()
'Asia'
>>> df['Continent'].max()
'America'

>>> df['Continent'].sort_values()
Company
Sony                      Asia
Lenovo                    Asia
Panasonic                 Asia
Tencent                   Asia
Hitachi                   Asia
LG Electronics            Asia
Foxconn                   Asia
Samsung Electronics       Asia
Huawei                    Asia
Dell Technologies      America
Facebook               America
HP Inc.                America
Microsoft              America
Intel                  America
IBM                    America
Alphabet               America
Apple                  America
```

(continué en la próxima página)

(proviene de la página anterior)

```
Name: Continent, dtype: category
Categories (5, object): ['Asia' < 'Africa' < 'Europe' < 'America' < 'Australia']
```

**Atención:** En condiciones normales (categorías sin ordenar) el mínimo hubiera sido America y el máximo hubiera sido Asia ya que se habrían ordenado alfabéticamente.

### Usando funciones estadísticas

Vamos a aplicar las funciones estadísticas que proporciona pandas sobre la columna **Revenue** de nuestro «dataset», aunque podríamos hacerlo sobre todas aquellas variables numéricas susceptibles:

```
>>> df['Revenue']
Company
Apple                  274515
Samsung Electronics    200734
Alphabet               182527
Foxconn                181945
Microsoft              143015
Huawei                 129184
Dell Technologies       92224
Facebook                85965
Sony                    84893
Hitachi                 82345
Intel                   77867
IBM                     73620
Tencent                 69864
Panasonic               63191
Lenovo                  60742
HP Inc.                 56639
LG Electronics          53625
Name: Revenue, dtype: int64
```

Tabla 3: Funciones estadísticas en pandas

| Función | Resultado | Descripción |
|---|---|---|
| `df['Revenue'].count()` | 17 | Número de observaciones no nulas |
| `df['Revenue'].sum()` | 1912895 | Suma de los valores |
| `df['Revenue'].mean()` | 112523.23 | Media de los valores |
| `df['Revenue'].mad()` | 51385.95 | Desviación absoluta media |

continué en la próxima página

Tabla 3 – proviene de la página anterior

| Función | Resultado | Descripción |
|---|---|---|
| `df['Revenue'].median()` | 84893.0 | Mediana de los valores |
| `df['Revenue'].min()` | 53625 | Mínimo |
| `df['Revenue'].max()` | 274515 | Máximo |
| `df['Revenue'].mode()` | Múltiples valores | Moda |
| `df['Revenue'].abs()` | Múltiples valores | Valor absoluto |
| `df['Revenue'].prod()` | 675806477941221376 | Producto de los valores |
| `df['Revenue'].std()` | 63236.95 | Desviación típica |
| `df['Revenue'].var()` | 3998912818.06 | Varianza |
| `df['Revenue'].sem()` | 15337.21 | Error típico de la media |
| `df['Revenue'].skew()` | 1.33 | Asimetría |
| `df['Revenue'].kurt()` | 1.13 | Apuntamiento |
| `df['Revenue'].quantile()` | 84893.0 | Cuantiles (por defecto 50%) |
| `df['Revenue'].cumsum()` | Múltiples valores | Suma acumulativa |
| `df['Revenue'].cumprod()` | Múltiples valores | Producto acumulativo |
| `df['Revenue'].cummax()` | Múltiples valores | Máximo acumulativo |
| `df['Revenue'].cummin()` | Múltiples valores | Mínimo acumulativo |

**Ejercicio**

Partiendo del conjunto de datos `democan`, obtenga aquellas islas cuya población está por encima de la media del archipiélago canario.

Resultado esperado: `['Gran Canaria', 'Tenerife']`

## Ordenando valores

Una operación muy típica cuando trabajamos con datos es la de ordenarlos en base a ciertos criterios. Veamos cómo podemos hacerlo utilizando pandas. Volvemos a nuestro «dataset» tecnológico:

```
>>> df
                        Revenue  Employees              City         Country
```

(continué en la próxima página)

(proviene de la página anterior)

```
Company
Apple                 274515     147000       California  United States
Samsung Electronics   200734     267937            Suwon    South Korea
Alphabet              182527     135301       California  United States
Foxconn               181945     878429  New Taipei City         Taiwan
Microsoft             143015     163000       Washington  United States
Huawei                129184     197000         Shenzhen          China
Dell Technologies      92224     158000            Texas  United States
Facebook               85965      58604       California  United States
Sony                   84893     109700            Tokyo          Japan
Hitachi                82345     350864            Tokyo          Japan
Intel                  77867     110600       California  United States
IBM                    73620     364800         New York  United States
Tencent                69864      85858         Shenzhen          China
Panasonic              63191     243540            Osaka          Japan
Lenovo                 60742      71500        Hong Kong          China
HP Inc.                56639      53000       California  United States
LG Electronics         53625      75000            Seoul    South Korea
```

Supongamos que queremos tener el conjunto de datos **ordenado por el nombre de empresa**. Como, en este caso, la columna Company constituye el índice, debemos ordenar por el índice:

```
>>> df.sort_index()
                     Revenue  Employees             City        Country
Company
Alphabet              182527     135301       California  United States
Apple                 274515     147000       California  United States
Dell Technologies      92224     158000            Texas  United States
Facebook               85965      58604       California  United States
Foxconn               181945     878429  New Taipei City         Taiwan
HP Inc.                56639      53000       California  United States
Hitachi                82345     350864            Tokyo          Japan
Huawei                129184     197000         Shenzhen          China
IBM                    73620     364800         New York  United States
Intel                  77867     110600       California  United States
LG Electronics         53625      75000            Seoul    South Korea
Lenovo                 60742      71500        Hong Kong          China
Microsoft             143015     163000       Washington  United States
Panasonic              63191     243540            Osaka          Japan
Samsung Electronics   200734     267937            Suwon    South Korea
Sony                   84893     109700            Tokyo          Japan
Tencent                69864      85858         Shenzhen          China
```

Ahora imaginemos que necesitamos tener las **empresas ordenadas de mayor a menor número de ingresos**:

```
>>> df.sort_values(by='Revenue', ascending=False)
                     Revenue  Employees            City        Country
Company
Apple                 274515     147000      California  United States
Samsung Electronics   200734     267937           Suwon    South Korea
Alphabet              182527     135301      California  United States
Foxconn               181945     878429  New Taipei City         Taiwan
Microsoft             143015     163000      Washington  United States
Huawei                129184     197000        Shenzhen          China
Dell Technologies      92224     158000           Texas  United States
Facebook               85965      58604      California  United States
Sony                   84893     109700           Tokyo          Japan
Hitachi                82345     350864           Tokyo          Japan
Intel                  77867     110600      California  United States
IBM                    73620     364800        New York  United States
Tencent                69864      85858        Shenzhen          China
Panasonic              63191     243540           Osaka          Japan
Lenovo                 60742      71500       Hong Kong          China
HP Inc.                56639      53000      California  United States
LG Electronics         53625      75000           Seoul    South Korea
```

También es posible utilizar varias columnas en la ordenación. Pongamos que deseamos **ordenar los datos por país y por ciudad**. Veamos cómo afrontarlo:

```
>>> df.sort_values(by=['Country', 'City'])
                     Revenue  Employees            City        Country
Company
Lenovo                 60742      71500       Hong Kong          China
Huawei                129184     197000        Shenzhen          China
Tencent                69864      85858        Shenzhen          China
Panasonic              63191     243540           Osaka          Japan
Sony                   84893     109700           Tokyo          Japan
Hitachi                82345     350864           Tokyo          Japan
LG Electronics         53625      75000           Seoul    South Korea
Samsung Electronics   200734     267937           Suwon    South Korea
Foxconn               181945     878429  New Taipei City         Taiwan
Apple                 274515     147000      California  United States
Alphabet              182527     135301      California  United States
Facebook               85965      58604      California  United States
Intel                  77867     110600      California  United States
HP Inc.                56639      53000      California  United States
IBM                    73620     364800        New York  United States
Dell Technologies      92224     158000           Texas  United States
Microsoft             143015     163000      Washington  United States
```

### Buscando máximos y mínimos

Al igual que veíamos *en el caso de las series*, podemos aplicar muchas de estas funciones de máximos y mínimos sobre un DataFrame de Pandas.

Podemos obtener los **valores mínimos y máximos de todas las columnas**:

```
>>> df.min()
Revenue            53625
Employees          53000
City          California
Country            China
dtype: object

>>> df.max()
Revenue              274515
Employees            878429
City             Washington
Country       United States
dtype: object
```

También podría ser de utilidad saber **qué empresa tiene el valor mínimo o máximo** para una determinada columna:

```
# LG tiene los menores ingresos
>>> df['Revenue'].idxmin()
'LG Electronics'

# Foxconn tiene el mayor número de empleados/as
>>> df['Employees'].idxmax()
'Foxconn'
```

**Nota:** En este caso nos devuelve una cadena de texto con el nombre de la empresa ya que tenemos definido así nuestro índice (etiquetas). En otro caso devolvería la posición (numérica) con un índice por defecto.

Si queremos acceder al registro completo, basta con acceder a través de la etiqueta devuelta:

```
>>> company = df['Revenue'].idxmin()

>>> df.loc[company]
Revenue             53625
Employees           75000
City                Seoul
Country       South Korea
```

(continué en la próxima página)

(proviene de la página anterior)

```
Name: LG Electronics, dtype: object
```

Otra de las operaciones muy usuales es encontrar los $n$ registros con mayores/menores valores. Supongamos que nos interesa conocer las **3 empresas con mayores ingresos y las 3 empresas con menor número de empleados/as**:

```
>>> df['Revenue'].nlargest(3)
Company
Apple                  274515
Samsung Electronics    200734
Alphabet               182527
Name: Revenue, dtype: int64

>>> df['Employees'].nsmallest(3)
Company
HP Inc.     53000
Facebook    58604
Lenovo      71500
Name: Employees, dtype: int64
```

**Nota:** Si no especificamos un número de registros, estas funciones lo tienen definido por defecto a 5.

Si queremos acceder al registro completo, podemos aplicar estas funciones de otro modo:

```
>>> df.nlargest(3, 'Revenue')
                     Revenue  Employees        City        Country
Company
Apple                 274515     147000  California  United States
Samsung Electronics   200734     267937       Suwon    South Korea
Alphabet              182527     135301  California  United States

>>> df.nsmallest(3, 'Employees')
          Revenue  Employees        City        Country
Company
HP Inc.     56639      53000  California  United States
Facebook    85965      58604  California  United States
Lenovo      60742      71500   Hong Kong          China
```

**Ejercicio**

Partiendo del conjunto de datos `democan` obtenga las 3 islas con menor densidad de población.

El resultado debería ser el siguiente:

```
              Population     Area Province    Density
Island
El Hierro          11147   278.71     SCTF  39.994977
La Gomera          21678   369.76     SCTF  58.627218
Fuerteventura     119732  1659.00     LPGC  72.171187
```

### Gestionando valores nulos

La limpieza de un «dataset» suele estar vinculado, en muchas ocasiones, a la gestión de los valores nulos. En este sentido, pandas ofrece varias funciones.

Para ejemplificar este apartado, vamos a hacer uso del siguiente DataFrame:

```
>>> df
   A    B    C
0  1  4.0  7.0
1  2  NaN  8.0
2  3  6.0  NaN
```

Si queremos **detectar aquellos valores nulos**, haremos lo siguiente:

```
>>> df.isna()
       A      B      C
0  False  False  False
1  False   True  False
2  False  False   True
```

**Nota:** También existe la función `isnull()` que funciona de manera análoga a `isna()`. En StackExchange puedes ver una explicación de estas funciones.

En caso de que nos interese **descartar los registros con valores nulos**, procedemos así:

```
>>> df.dropna()
   A    B    C
0  1  4.0  7.0
```

Sin embargo, también existe la posibilidad de **rellenar los valores nulos** con algún sustituto. En este caso podemos ejecutar lo siguiente:

```
>>> df.fillna(0)
   A    B    C
0  1  4.0  0.0
```

(continué en la próxima página)

(proviene de la página anterior)

```
1  2  0.0  0.0
2  3  6.0  9.0
```

Incluso podemos **aplicar interpolación para completar valores nulos**:

```
>>> df.interpolate()
   A    B    C
0  1  4.0  7.0
1  2  5.0  8.0
2  3  6.0  8.0
```

### Reformando datos

En esta sección se verán las operaciones de **pivotar** y **apilar** que permiten reformar (remodelar) un DataFrame.

Seguimos utilizando el conjunto de datos de empresas tecnológicas aunque nos quedaremos únicamente con las 3 primeras filas a efectos didácticos:

```
>>> df = df.reset_index()[:3]

>>> df
              Company  Revenue  Employees        City        Country
0               Apple   274515     147000  California  United States
1  Samsung Electronics   200734     267937       Suwon    South Korea
2            Alphabet   182527     135301  California  United States
```

#### Ancho y Largo

Típicamente existen dos maneras de presentar datos tabulares: formato ancho y formato largo. En **formato ancho** cada fila tiene múltiples columnas representando todas las variables de una misma observación. En **formato largo** cada fila tiene básicamente tres columnas: una que identifica la observación, otra que identifica la variable y otra que contiene el valor.

Para pasar de formato ancho a formato largo usamos la función melt():

```
>>> df.melt(id_vars='Company')
                Company   variable          value
0                 Apple    Revenue         274515
1   Samsung Electronics    Revenue         200734
2              Alphabet    Revenue         182527
```

(continué en la próxima página)

(proviene de la página anterior)

```
3                  Apple  Employees         147000
4    Samsung Electronics  Employees         267937
5               Alphabet  Employees         135301
6                  Apple       City     California
7    Samsung Electronics       City          Suwon
8               Alphabet       City     California
9                  Apple    Country  United States
10   Samsung Electronics    Country    South Korea
11              Alphabet    Country  United States
```

Para pasar de formato largo a formato ancho usamos la función pivot():

```
>>> df_long = df.melt(id_vars='Company')

>>> df_long.pivot(index='Company', columns='variable', values='value')
variable                   City        Country Employees Revenue
Company
Alphabet             California  United States    135301  182527
Apple                California  United States    147000  274515
Samsung Electronics       Suwon    South Korea    267937  200734
```

---

**Truco:** Nótese que las columnas tienen un nombre `variable` que se puede modificar mediante `columns.name`.

---

Si queremos obtener el DataFrame en formato ancho, tal y como estaba, tenemos que realizar alguna operación adicional: `df.rename_axis(columns = None).reset_index()`.

### Apilando datos

Las operaciones de apilado trabajan sobre los índices del DataFrame. Para comprobar su aplicabilidad, vamos a añadir la columna «Company» como índice del «dataset» anterior:

```
>>> df.set_index('Company', inplace=True)

>>> df
                     Revenue  Employees        City        Country
Company
Apple                 274515     147000  California  United States
Samsung Electronics   200734     267937       Suwon    South Korea
Alphabet              182527     135301  California  United States
```

La función stack() nos permite obtener un DataFrame con **índice multinivel** que incluye las columnas del DataFrame de origen y los valores agrupados:

```
>>> df_stacked = df.stack()

>>> df_stacked
Company
Apple                Revenue             274515
                     Employees           147000
                     City            California
                     Country      United States
Samsung Electronics  Revenue             200734
                     Employees           267937
                     City                 Suwon
                     Country        South Korea
Alphabet             Revenue             182527
                     Employees           135301
                     City            California
                     Country      United States
dtype: object

>>> df_stacked.index
MultiIndex([(              'Apple',   'Revenue'),
            (              'Apple', 'Employees'),
            (              'Apple',      'City'),
            (              'Apple',   'Country'),
            ('Samsung Electronics',   'Revenue'),
            ('Samsung Electronics', 'Employees'),
            ('Samsung Electronics',      'City'),
            ('Samsung Electronics',   'Country'),
            (           'Alphabet',   'Revenue'),
            (           'Alphabet', 'Employees'),
            (           'Alphabet',      'City'),
            (           'Alphabet',   'Country')],
           names=['Company', None])
```

La función unstack() realiza justo la operación contraria: convertir un DataFrame con índice multinivel en un Dataframe en formato ancho con índice sencillo. Se podría ver como una manera de **aplanar** el «dataset»:

```
>>> df_flat = df_stacked.unstack()

>>> df_flat
                    Revenue Employees        City        Country
Company
Apple                274515    147000  California  United States
Samsung Electronics  200734    267937       Suwon    South Korea
Alphabet             182527    135301  California  United States
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> df_flat.index
Index(['Apple', 'Samsung Electronics', 'Alphabet'], dtype='object', name='Company')
```

## Agrupando datos

Las operaciones de agregado son muy recurridas y nos permiten extraer información relevante, que, a simple vista, quizás no sea tan evidente.

Veamos un ejemplo en el que calculamos la **suma de los ingresos de las empresas, agrupados por país**:

```
>>> df.groupby('Country')['Revenue'].sum()
Country
China            259790
Japan            230429
South Korea      254359
Taiwan           181945
United States    986372
Name: Revenue, dtype: int64
```

También es posible realizar la agrupación en varios niveles. En el siguiente ejemplo tendremos los datos **agrupados por país y ciudad**:

```
>>> df.groupby(['Country', 'City'])['Revenue'].sum()
Country        City
China          Hong Kong          60742
               Shenzhen          199048
Japan          Osaka              63191
               Tokyo             167238
South Korea    Seoul              53625
               Suwon             200734
Taiwan         New Taipei City   181945
United States  California        677513
               New York           73620
               Texas              92224
               Washington        143015
Name: Revenue, dtype: int64
```

**Ver también:**

Cuando realizamos una agrupación por varias columnas, el resultado contiene un índice de múltiples niveles. Podemos aplanar el DataFrame usando *unstack()*.

Incluso podemos aplicar distintas funciones de agregación a cada columna. Supongamos que necesitamos calcular **la media de los ingresos y la mediana del número de**

**empleados/as, con las empresas agrupadas por país**:

```
>>> df.groupby('Country').agg({'Revenue': 'mean', 'Employees': 'median'})
                    Revenue  Employees
Country
China          86596.666667    85858.0
Japan          76809.666667   243540.0
South Korea   127179.500000   171468.5
Taiwan        181945.000000   878429.0
United States  123296.500000   141150.5
```

**Nota:** Utilizamos la función `agg()` pasando un diccionario cuyas claves son nombres de columnas y cuyos valores son funciones a aplicar.

**Ejercicio**

Obtenga el porcentaje de población (en relación con el total) de cada provincia de las Islas Canarias en base al «dataset» `democan`.

El resultado debería ser similar a:

- Las Palmas de Gran Canaria: 52%
- Santa Cruz de Tenerife: 48%

### Aplicando funciones

Pandas permite la aplicación de funciones (tanto propias como «built-in») a filas y/o columnas de un DataFrame.

Numpy nos ofrece una amplia gama de funciones matemáticas. Podemos hacer uso de cualquier de ellas aplicándola directamente a nuestro conjunto de datos. Veamos un ejemplo en el que obtenemos **el máximo de cada columna**:

```
>>> df.apply(np.max)
Revenue              274515
Employees            878429
City             Washington
Country       United States
dtype: object
```

**Truco:** En este caso equivalente a `df.max()`.

Podemos aplicar funciones sobre determinadas columnas. Supongamos que queremos obtener **el logaritmo de la serie de ingresos**:

```
>>> df['Revenue'].apply(np.log)
Company
Apple                  12.522761
Samsung Electronics    12.209736
Alphabet               12.114653
Foxconn                12.111460
Microsoft              11.870705
Huawei                 11.768993
Dell Technologies      11.431976
Facebook               11.361696
Sony                   11.349147
Hitachi                11.318673
Intel                  11.262758
IBM                    11.206672
Tencent                11.154306
Panasonic              11.053917
Lenovo                 11.014391
HP Inc.                10.944453
LG Electronics         10.889771
Name: Revenue, dtype: float64
```

Ahora imaginemos un escenario en el que **la normativa de Estados Unidos ha cambiado y obliga a sus empresas tecnológicas a aumentar un 5% el número de empleados/as** que tienen. Esto lo podríamos abordar escribiendo una función propia que gestione cada fila del «dataset» y devuelva el valor adecuado de empleados/as según las características de cada empresa:

```
>>> def raise_employment(row):
...     num_employees = row['Employees']
...     if row['Country'] == 'United States':
...         return num_employees * 1.05
...     return num_employees
```

Ahora ya podemos aplicar esta función a nuestro DataFrame, teniendo en cuenta que debemos actuar sobre el **eje de filas** (`axis=1`):

```
>>> df.apply(raise_employment, axis=1)
Company
Apple                  154350.00
Samsung Electronics    267937.00
Alphabet               142066.05
Foxconn                878429.00
Microsoft              171150.00
Huawei                 197000.00
```

(continué en la próxima página)

(proviene de la página anterior)

```
Dell Technologies      165900.00
Facebook                61534.20
Sony                   109700.00
Hitachi                350864.00
Intel                  116130.00
IBM                    383040.00
Tencent                 85858.00
Panasonic              243540.00
Lenovo                  71500.00
HP Inc.                 55650.00
LG Electronics          75000.00
dtype: float64
```

El resultado es una serie que se podría incorporar al conjunto de datos, o bien, reemplazar la columna *Employees* con estos valores.

---

**Ejercicio**

Supongamos que el Gobierno de Canarias va a dar unas ayudas a cada isla en función de su superficie y su población, con las siguientes reglas:

- Islas con menos de 1000 $km^2$: ayuda del 30% de su población.
- Islas con más de 1000 $km^2$: ayuda del 20% de su población.

Añada una nueva columna *Grant* al «dataset» `democan` donde se contemplen estas ayudas. El DataFrame debería quedar así:

```
               Population     Area Province     Grant
Island
Gran Canaria       855521  1560.10     LPGC  171104.2
Tenerife           928604  2034.38     SCTF  185720.8
La Palma            83458   708.32     SCTF   25037.4
Lanzarote          155812   845.94     LPGC   46743.6
La Gomera           21678   369.76     SCTF    6503.4
El Hierro           11147   278.71     SCTF    3344.1
Fuerteventura      119732  1659.00     LPGC   23946.4
```

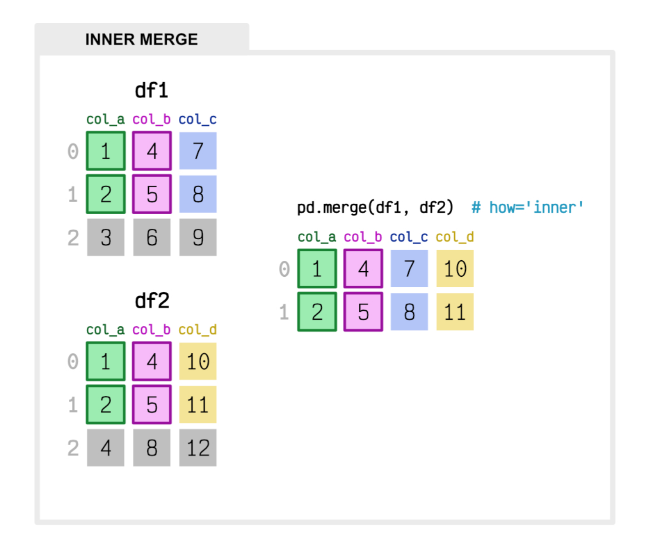
---

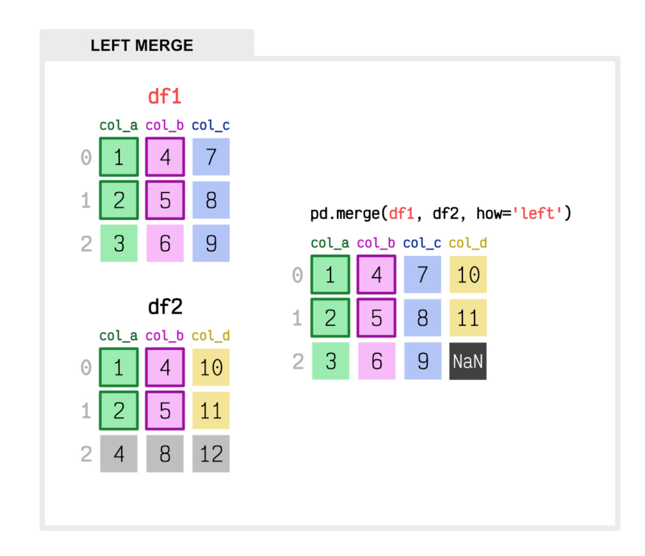
### Uniendo DataFrames

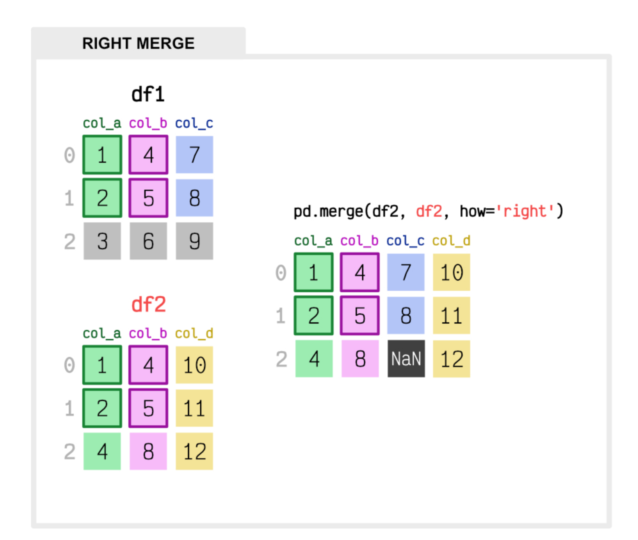
En esta sección veremos dos técnicas: Una de ellas «fusiona» dos DataFrames mientras que la otra los «concatena».

#### Fusión de DataFrames

Pandas proporciona la función merge() para mezclar dos DataFrames. El comportamiento de la función viene definido, entre otros, por el parámetro `how` que establece el método de «fusión»:



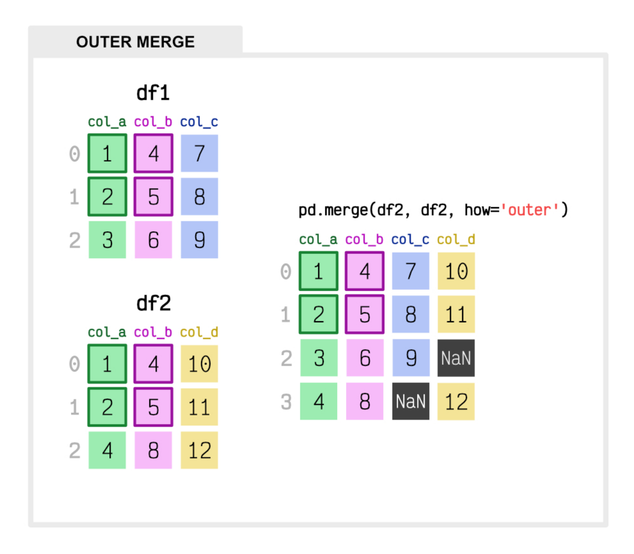




Figura 19: Operaciones de mezcla con «merge»

En principio, si no establecemos ningún argumento adicional, «merge» tratará de vincular aquellas filas con columnas homónimas en ambos conjuntos de datos. Si queremos especificar

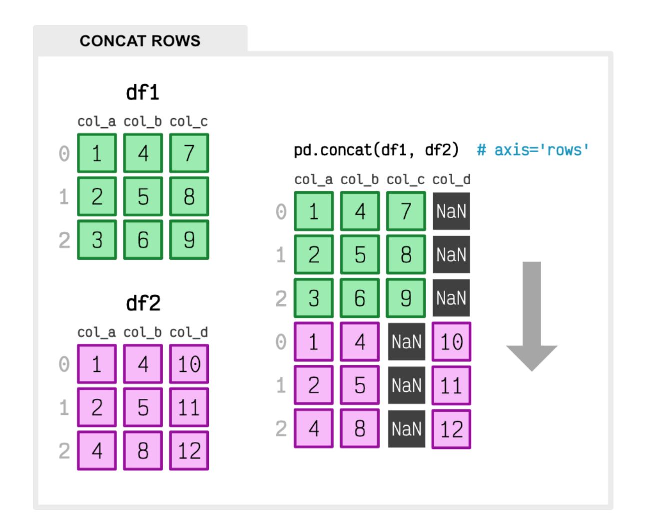
que la mezcla se dirija por determinadas columnas, tenemos a disposición los parámetros `on`, `left_on` o `right_on`.

**Ver también:**

Existe la posibilidad de generar un producto cartesiano entre las filas de ambos DataFrames. Para ello podemos usar `pd.merge(df1, df2, how='cross')`.

## Concatenación de DataFrames

Para concatenar dos DataFrames podemos utilizar la función concat() que permite añadir las filas de un DataFrame a otro, o bien añadir las columnas de un DataFrame a otro.



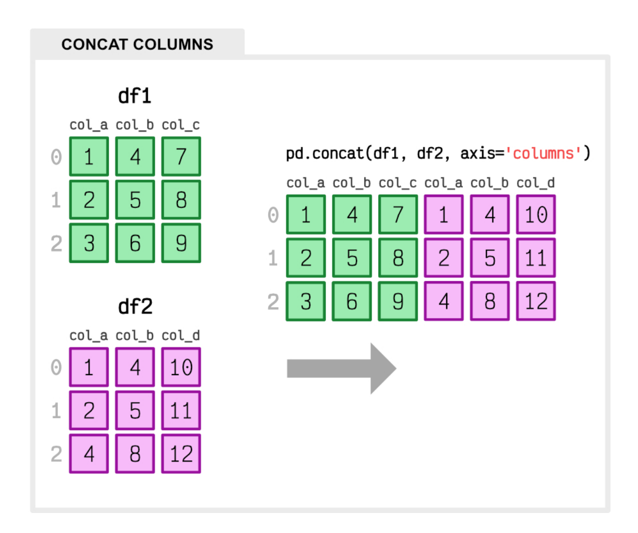




Figura 20: Operaciones de concatenación con «concat»

Si queremos «reindexar» el DataFrame concatenado, la función `concat()` admite un parámetro `ignore_index` que podemos poner a `True`. De esta forma tendremos un «dataset» resultante con índice desde 0 hasta N.

---

**Ejercicio**

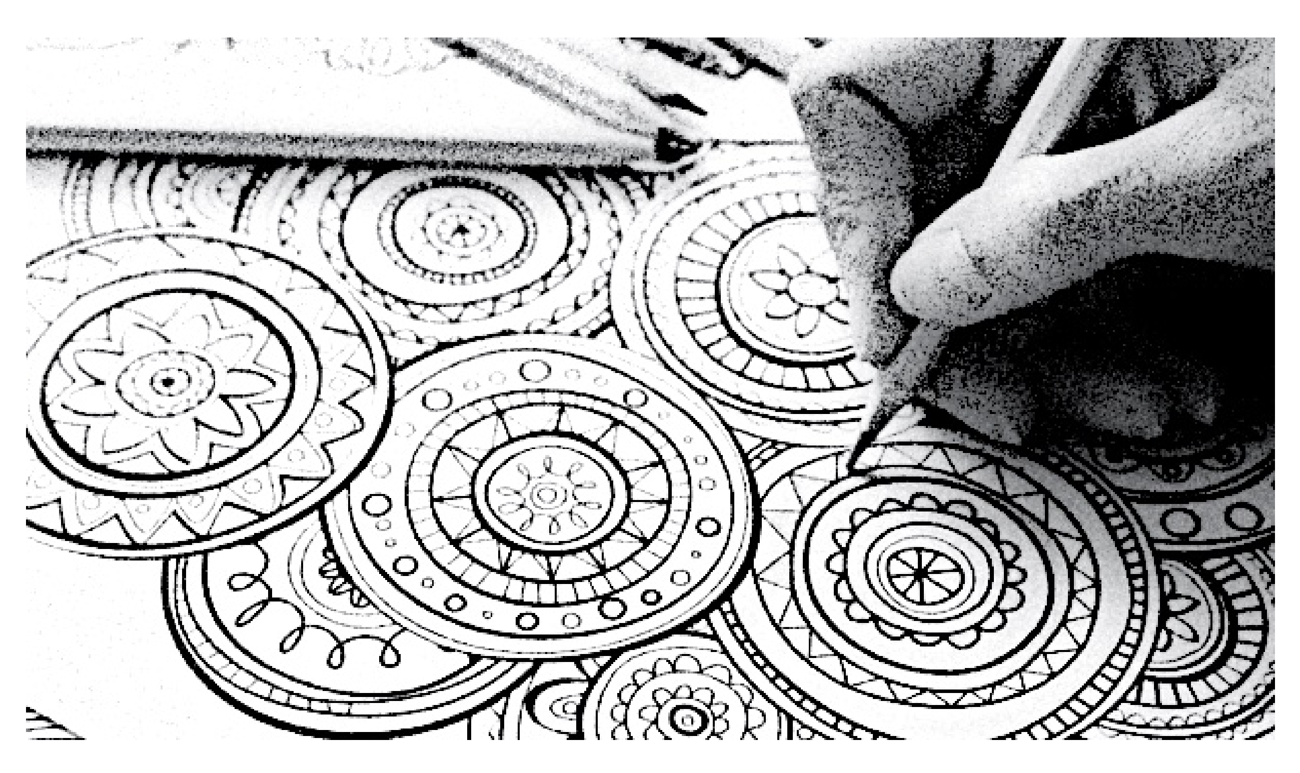
Obtenga los datos de población y superficie de las comunidades autónomas españolas desde esta url de Wikipedia en un único DataFrame con la siguiente estructura:

```
            Comunidad  Superficie  Población    Densidad
0     Castilla y León       94226    2407650   25.551865
1           Andalucía       87268    8379248   96.017418
2   Casstilla-La Mancha     79463    2025510   25.489976
...
...
```

Notas:

- Utilice la función `pd.read_html()` para acceder a las tablas. La tabla de superficie tiene el índice 3 y la tabla de población tiene el índice 4.
- Elimine la última fila de totales en cada DataFrame y quédese sólo con las columnas que interesen.
- Renombre las columnas según interese.
- Reemplace los valores de población y superficie para que sean números y convierta las columnas a entero.
- Realice la mezcla de población y superficie en un único DataFrame.
- Calcule la densidad de población de cada comunidad autónoma.

## 9.4 matplotlib



matplotlib es el paquete Python más utilizado en el ámbito de la ciencia de datos para representaciones gráficas.[1]

```
$ pip install matplotlib
```

La forma más común de importar esta librería es usar el alias `plt` de la siguiente manera:

[1] Foto original de portada por Customerbox en Unsplash.

```
>>> import matplotlib.pyplot as plt
```

**Importante:** Si bien podemos utilizar matplotlib en el intérprete habitual de Python, suele ser muy frecuente trabajar con esta librería mediante entornos *Jupyter*, ya que facilitan la visualización de los gráficos en su interfaz de usuario.

### 9.4.1 Figura

La figura es el elemento base sobre el que se construyen todos los gráficos en matplotlib. Veamos cómo crearla:

```
>>> fig = plt.figure()

>>> type(fig)
matplotlib.figure.Figure

>>> fig
<Figure size 640x480 with 0 Axes>
```

Podemos observar que la resolución (por defecto) de la figura es de 640x480 píxeles y que no dispone de ningún eje («0 Axes»).

**Importante:** El término «axes» hace referencia a un conjunto de ejes. Puede resultar confuso en español y he decidido asignar el nombre **marco** cuando haga referencia a «axes».

La resolución final de una figura viene determinada por su altura (`height`) y anchura (`width`) especificadas en pulgadas[2] que, a su vez, se multiplican por los puntos por pulgada o `dpi`. Veamos el funcionamiento:

```
>>> fig
<Figure size 640x480 with 0 Axes>

>>> fig.get_figwidth()   # pulgadas
6.4
>>> fig.get_figheight()  # pulgadas
4.8
>>> fig.get_dpi()        # dots per inch
100.0
```

(continué en la próxima página)

[2] Se suele usar el término inglés «inches».

(proviene de la página anterior)

```
>>> fig.get_figwidth() * fig.dpi, fig.get_figheight() * fig.dpi
(640.0, 480.0)
```

**Importante:** Si utilizamos entornos de desarollo basados en Jupyter, los valores por defecto son distintos:

- Ancho de figura: 6 in
- Alto de figura: 4 in
- DPI: 75
- Resolución: 450x300 px

Por tanto, cuando creamos una figura podemos modificar los parámetros por defecto para obtener la resolución deseada:

```
>>> fig = plt.figure(figsize=(19.2, 10.8))  # 100 dpi
>>> fig
<Figure size 1920x1080 with 0 Axes>

>>> fig = plt.figure(figsize=(19.2, 10.8), dpi=300)
>>> fig
<Figure size 5760x3240 with 0 Axes>
```

Si nos interesa que cualquier figura tome unos valores concretos de resolución, podemos modificar los **valores por defecto del entorno**. Para ello, matplotlib hace uso de un diccionario `plt.rcParams` que contiene los parámetros globales de configuración. Veamos cómo modificarlo:

```
>>> plt.rcParams['figure.figsize']
[6.4, 4.8]
>>> plt.rcParams['figure.dpi']
100.0

>>> plt.rcParams['figure.figsize'] = (10, 5)
>>> plt.rcParams['figure.dpi'] = 300          # res. final: 3000x1500 px

>>> fig.get_figwidth()
10.0
>>> fig.get_figheight()
5.0
>>> fig.dpi
300.0
```

### 9.4.2 Marcos

Para poder empezar a graficar necesitamos tener, al menos, un marco. Utilizaremos la función `add_subplot()` que requiere pasar como parámetros el número de filas, el número de columnas y el marco activo:

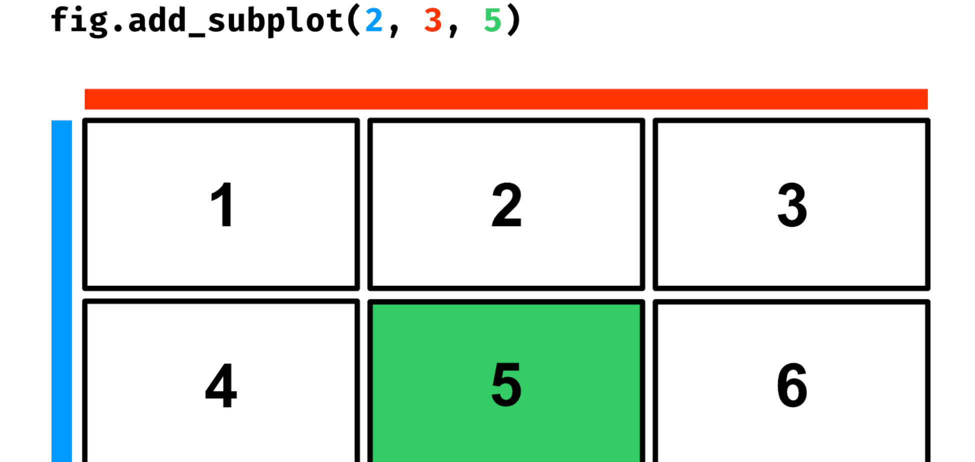



Figura 21: Creación de marcos dentro de una figura

Para comenzar vamos a trabajar únicamente con un marco:

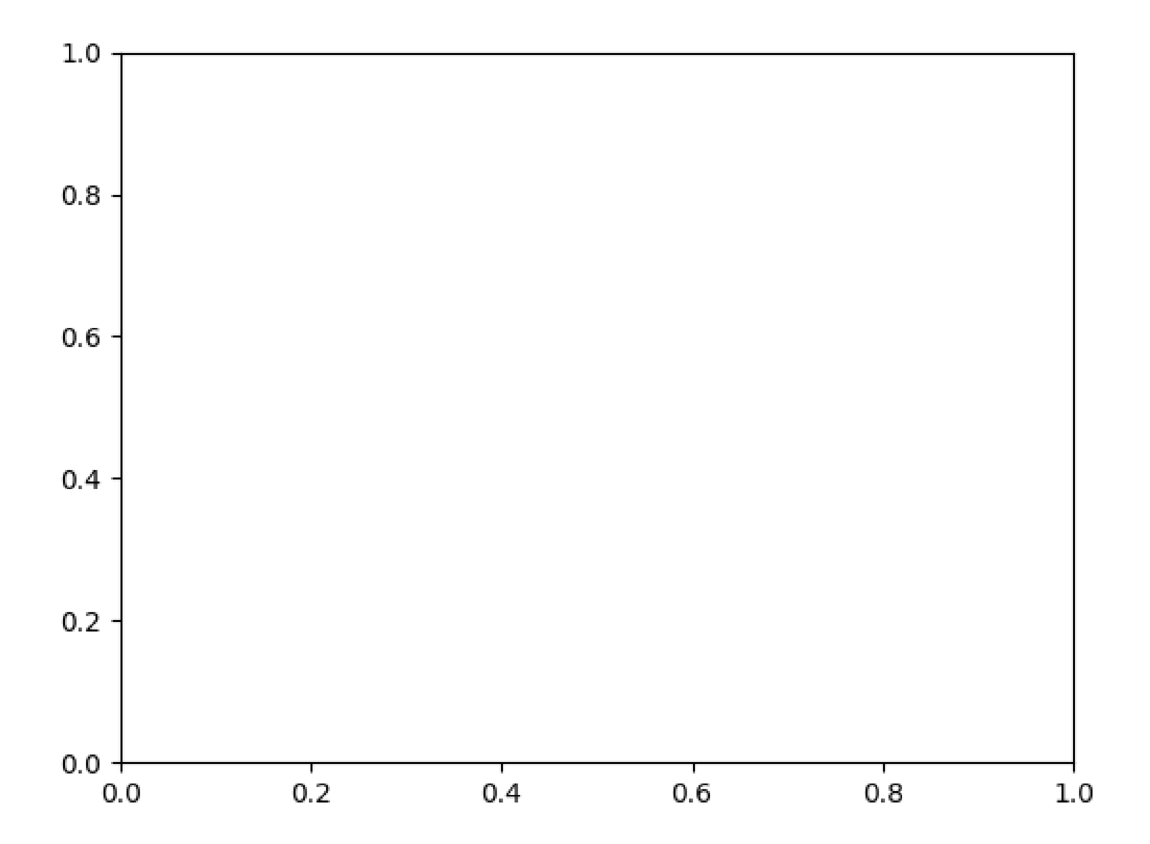
```
>>> fig = plt.figure()

>>> ax = fig.add_subplot(1, 1, 1)  # equivalente a fig.add_subplot(111)

>>> ax
<AxesSubplot:>

>>> fig
<Figure size 640x480 with 1 Axes>
```

**Truco:** Suele ser habitual encontrar `ax` como nombre de variable del «axes» devuelto por la función `add_subplot()`.




**Nota:** La escala por defecto de cada eje va de 0 a 1 con marcas cada 0.2

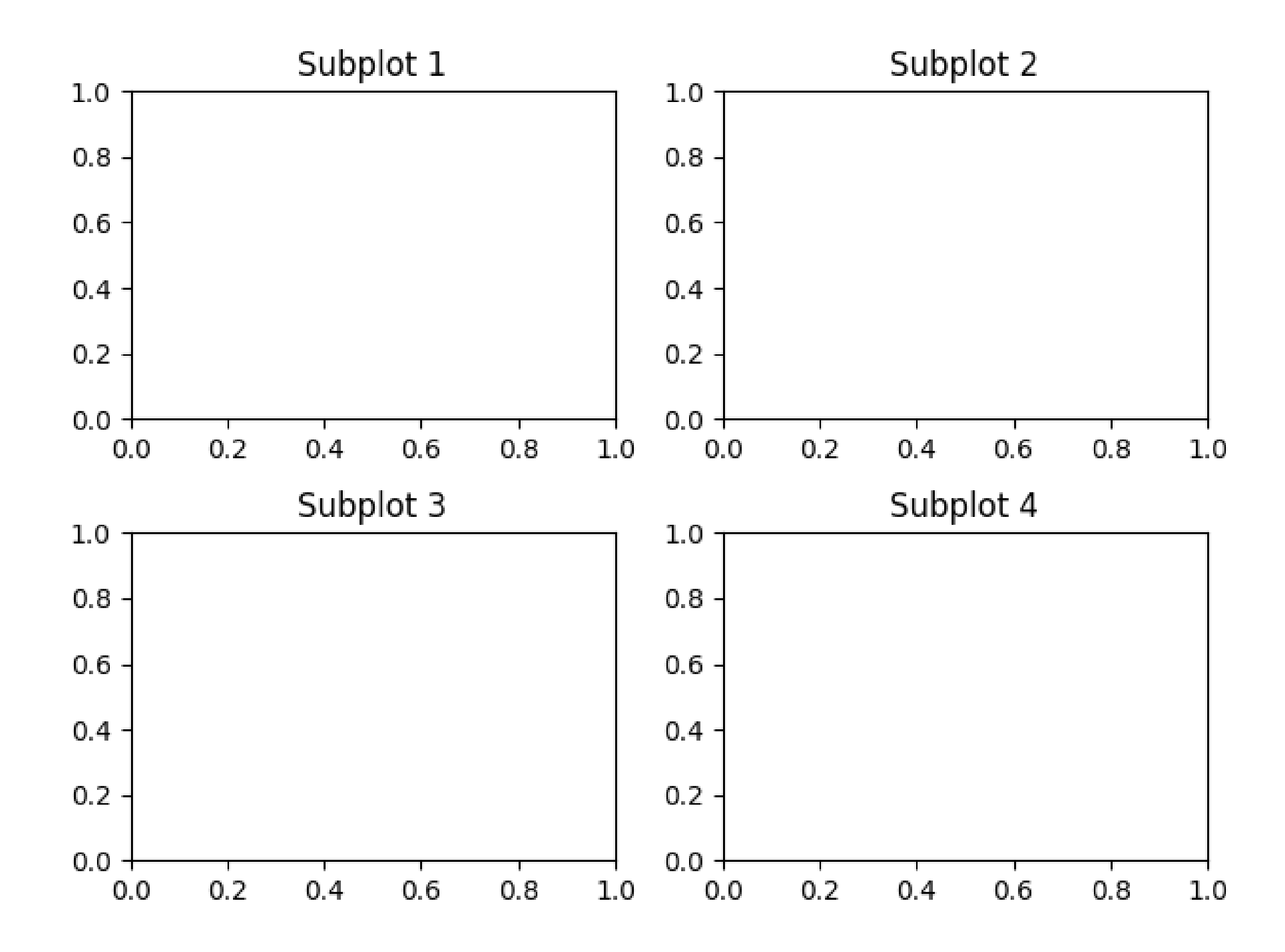
Ahora vamos a generar 4 marcos sobre los que fijaremos un título identificativo:

```
>>> fig = plt.figure()

>>> for i in range(1, 5):
...     ax = fig.add_subplot(2, 2, i)
...     ax.set_title(f'Subplot {i}')

>>> fig.tight_layout(pad=1)  # sólo para que no se solapen los títulos

>>> fig
<Figure size 640x480 with 4 Axes>
```




### Atajo para subgráficos

Matplotlib nos ofrece una forma compacta de crear a la vez tanto la **figura** como los **marcos** que necesitemos.

Para ello utilizaremos la función `plt.subplots()` que recibe como parámetros el *número de filas* y el *número de columnas* para la disposición de los marcos, y devuelve una tupla con la figura y los marcos.

En el siguiente ejemplo creamos **una figura con un único marco**:

```
>>> fig, ax = plt.subplots(1, 1)

>>> fig
<Figure size 640x480 with 1 Axes>

>>> ax
<AxesSubplot:>
```

**Truco:** Si invocamos la función `plt.subplots()` sin parámetros, creará (por defecto) un

único marco.

---

En el siguiente ejemplo creamos **una figura con 6 marcos** en disposición de 2 filas por 3 columnas:

```
>>> fig, ax = plt.subplots(2, 3)

>>> fig
<Figure size 640x480 with 6 Axes>

>>> ax
array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)

>>> ax.shape
(2, 3)
```

---

**Nota:** Se podría ver la función `subplots()` como una combinación de `figure()` + `add_subplot()`.

---

### Etiquetas

Dentro de un marco también es posible fijar las etiquetas de los ejes (X e Y). Veamos cómo hacerlo:

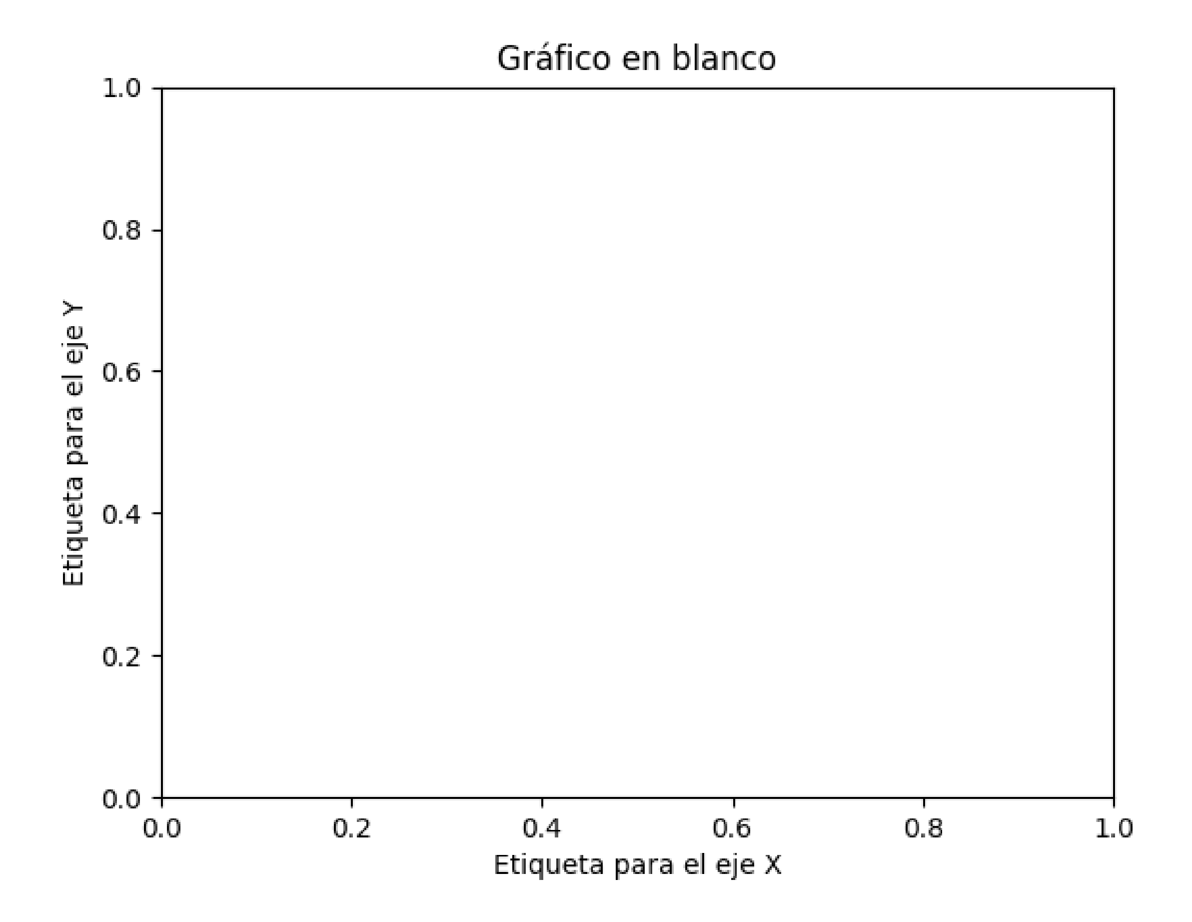
```
>>> fig, ax = plt.subplots()

>>> ax.set_title('Gráfico en blanco')
Text(0.5, 1.0, 'Gráfico en blanco')

>>> ax.set_xlabel('Etiqueta para el eje X')
Text(0.5, 0, 'Etiqueta para el eje X')

>>> ax.set_ylabel('Etiqueta para el eje Y')
Text(0, 0.5, 'Etiqueta para el eje Y')

>>> fig
<Figure size 640x480 with 1 Axes>
```




### Ejes

Un marco (2D) está compuesto por dos ejes: eje X e eje Y. Podemos acceder a cada eje mediante sendos atributos:

```
>>> ax.xaxis
<matplotlib.axis.XAxis at 0x112b34100>

>>> ax.yaxis
<matplotlib.axis.YAxis at 0x112b34850>
```

### Rejilla

En cada eje podemos activar o desactivar la rejilla, así como indicar su estilo.

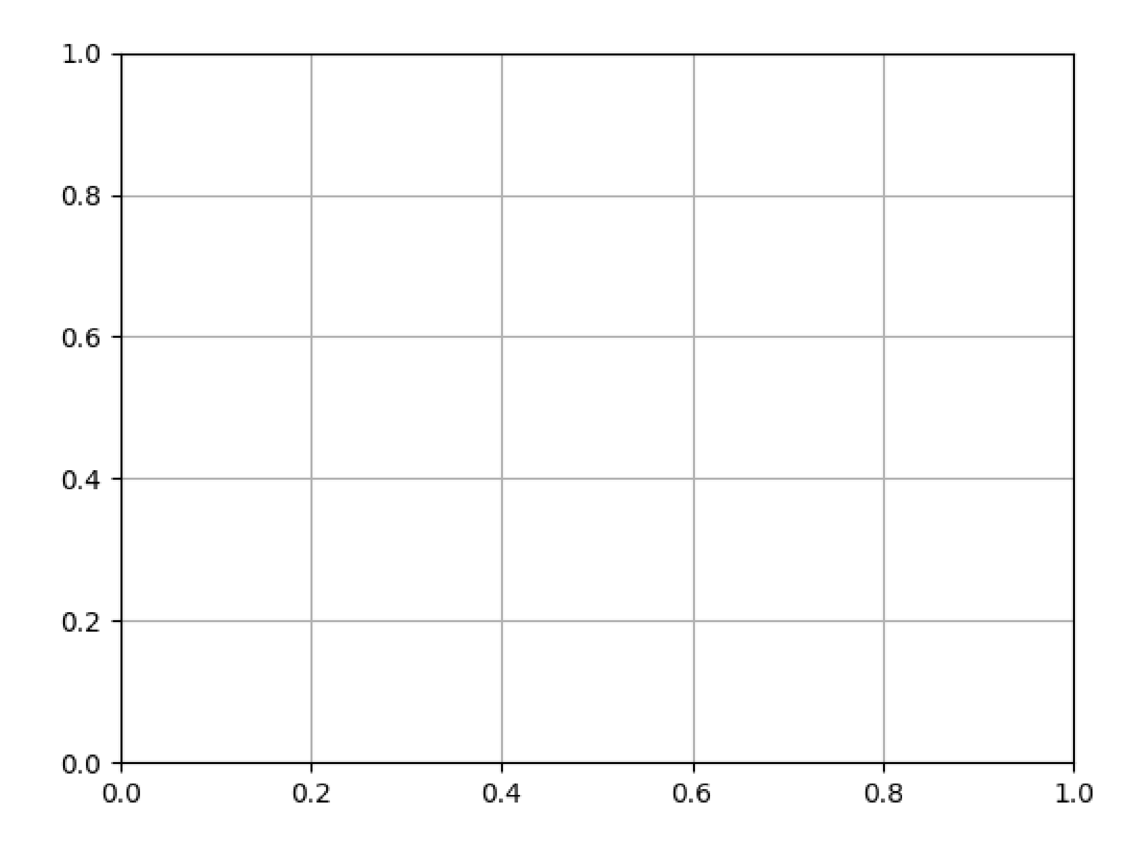
En primer lugar vamos a activar la rejilla en ambos ejes:

```
>>> ax.xaxis.grid(True)
>>> ax.yaxis.grid(True)
```

Esto sería equivalente a:

```
>>> ax.grid(True)
```

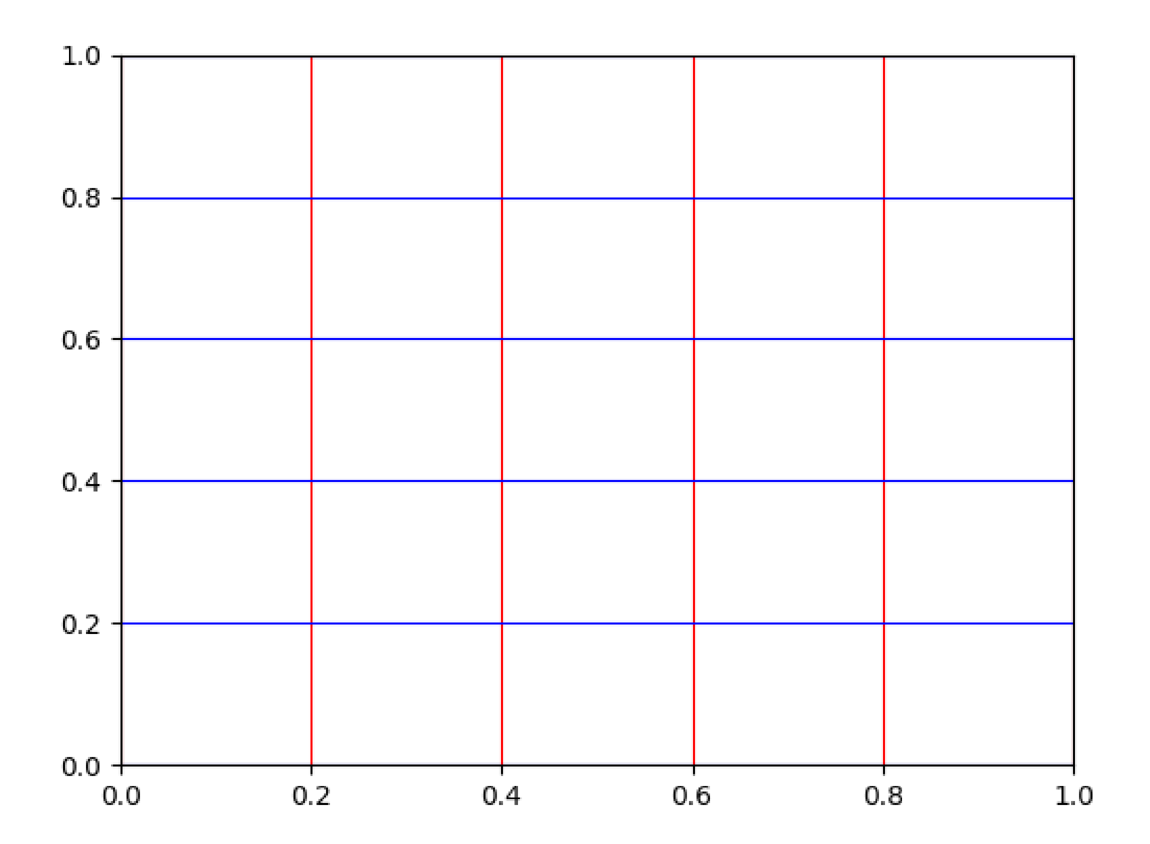
Y obtendríamos una figura con la rejilla (por defecto):




**Truco:** Las funciones de matplotlib que actúan como «interruptores» tienen por defecto el valor verdadero. En este sentido `ax.grid()` invocada sin parámetros hace que se muestre la rejilla. Esto se puede aplicar a muchas otras funciones.

Supongamos ahora que queremos personalizar la rejilla con **estilos diferentes en cada eje**:

```
>>> ax.xaxis.grid(color='r', linestyle='-')  # equivale a color='red',  linestyle=
↪'solid'
>>> ax.yaxis.grid(color='b', linestyle='-')  # equivale a color='blue', linestyle=
↪'solid'
```




- Parámetros disponibles para creación del grid.
- Listado de nombres de colores en matplotlib.
- Estilos de línea en matplotlib.

### Marcas

Por defecto, los ejes del marco tienen unas marcas[3] equiespaciadas que constituyen las *marcas mayores*. Igualmente existen unas *marcas menores* que, a priori, no están activadas.

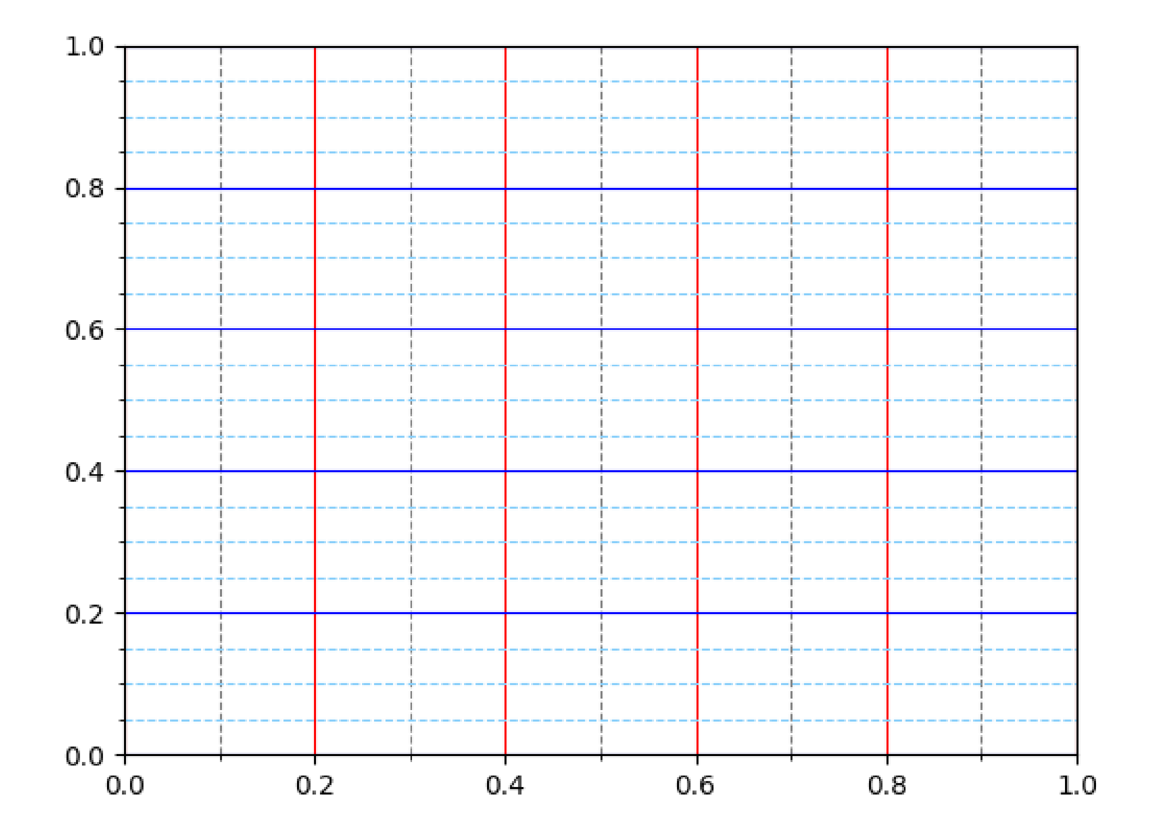
Ambos elementos son susceptibles de modificarse. Veamos un ejemplo en el que establecemos las **marcas menores con distinto espaciado en cada eje** y además le damos un estilo diferente a cada rejilla:

[3] Se suele usar el término inglés «ticks».

```
>>> from matplotlib.ticker import MultipleLocator

>>> ax.xaxis.set_minor_locator(MultipleLocator(0.1))   # X: separación cada 0.1␣
↪unidades
>>> ax.yaxis.set_minor_locator(MultipleLocator(0.05))  # Y: separación cada 0.05␣
↪unidades

>>> ax.xaxis.grid(which='minor', linestyle='dashed', color='gray')
>>> ax.yaxis.grid(which='minor', linestyle='dashed', color='lightskyblue')
```




También es posible asignar etiquetas a las marcas menores. En ese sentido, veremos un ejemplo en el que incorporamos los **valores a los ejes con estilos propios**:

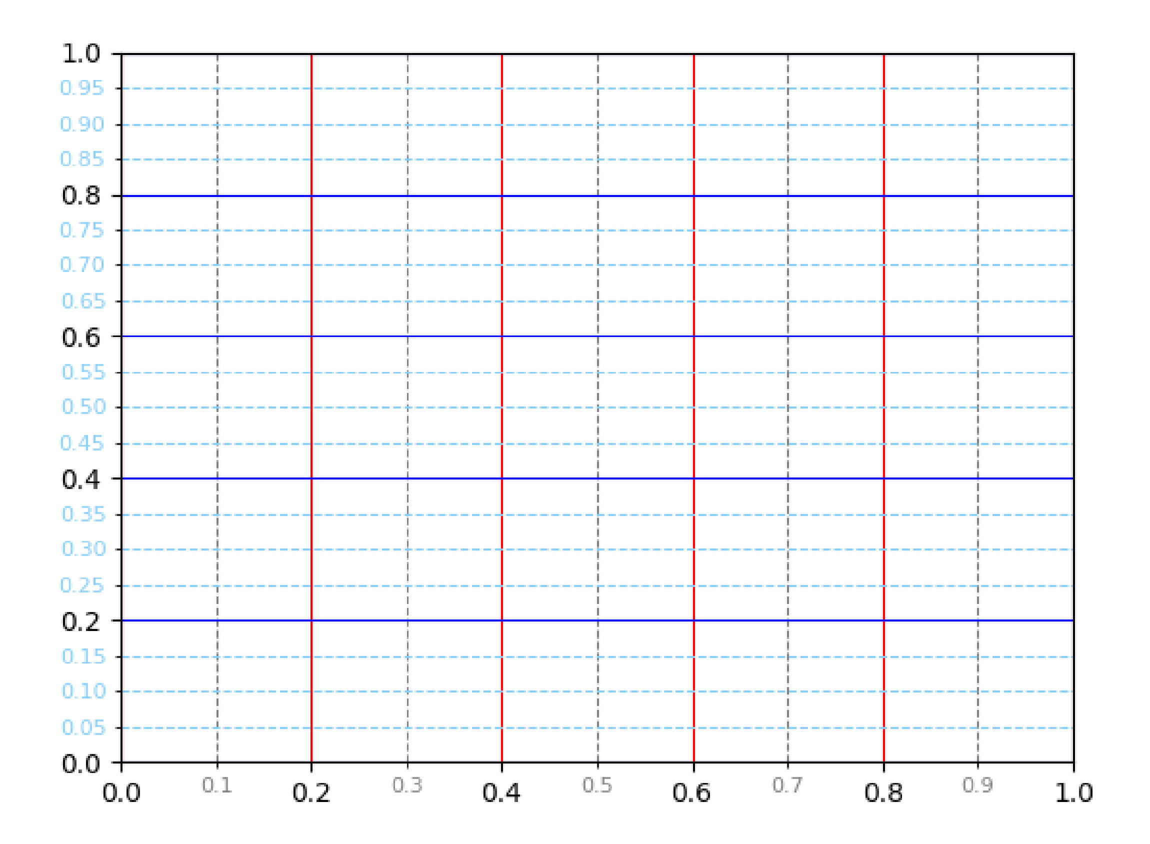
- Marcas menores en el eje X: precisión de 1 decimal, tamaño de letra 8 y color gris.
- Marcas menores en el eje Y: precisión de 2 decimales, tamaño de letra 8 y color azul.

```
>>> # Eje X
>>> ax.xaxis.set_minor_formatter('{x:.1f}')
>>> ax.tick_params(axis='x', which='minor', labelsize=8, labelcolor='gray')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> # Eje Y
>>> ax.yaxis.set_minor_formatter('{x:.2f}')
>>> ax.tick_params(axis='y', which='minor', labelsize=8, labelcolor='lightskyblue')
```

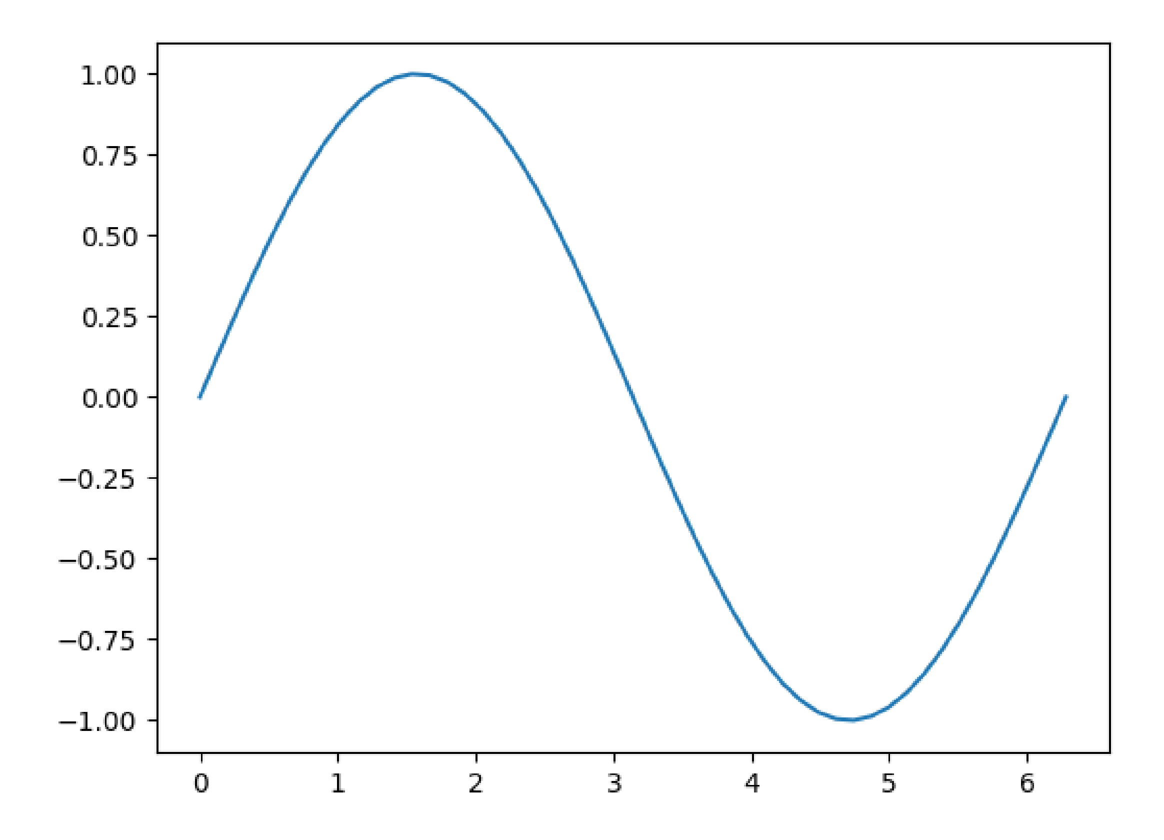



### 9.4.3 Primeros pasos

Vamos a empezar por representar la función $f(x) = sin(x)$. Para ello crearemos una variable $x$ con valores flotantes equidistantes y una variable $y$ aplicando la función senoidal. Nos apoyamos en *numpy* para ello. A continuación usaremos la función `plot()` del marco para representar la función creada:

```
>>> x = np.linspace(0, 2 * np.pi)
>>> y = np.sin(x)

>>> fig, ax = plt.subplots()
>>> ax.plot(x, y)
[<matplotlib.lines.Line2D at 0x120914040>]
```




### Múltiples funciones

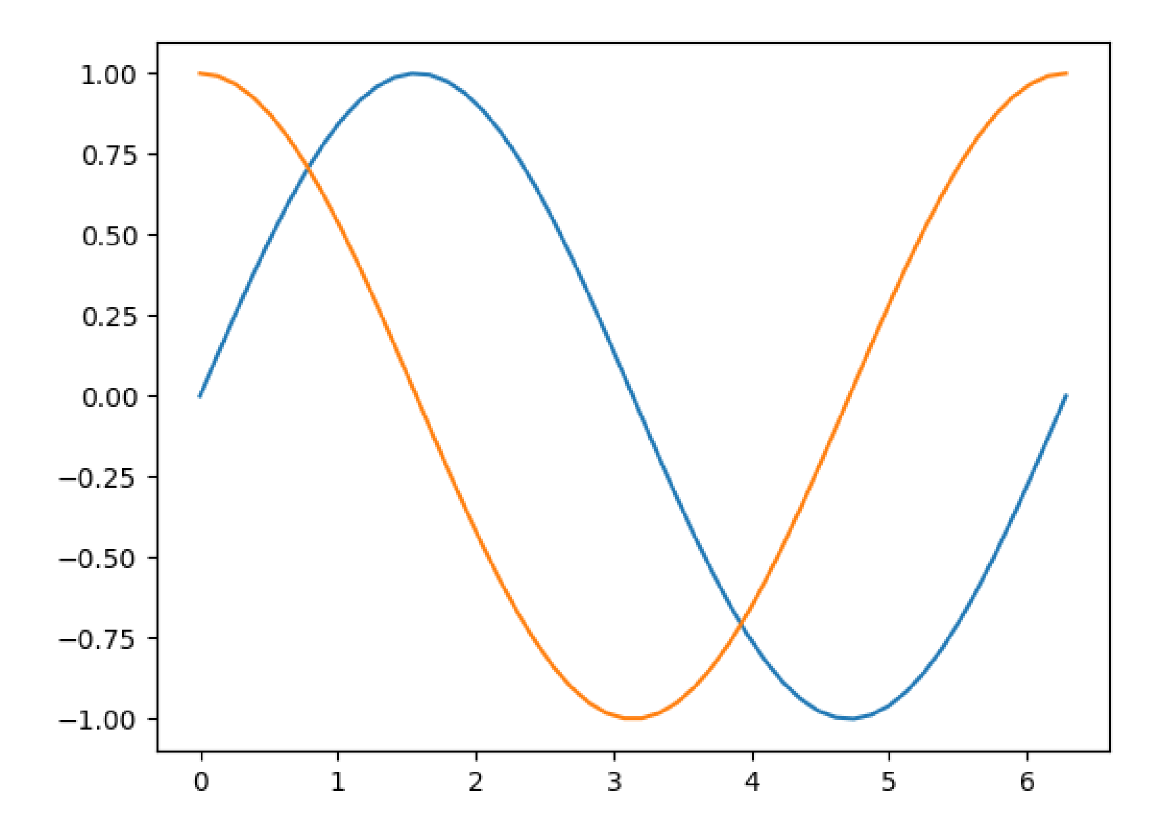
Partiendo de un mismo marco, es posible graficar todas las funciones que necesitemos. A continuación crearemos un marco con las funciones seno y coseno:

```
>>> x = np.linspace(0, 2 * np.pi)
>>> sin = np.sin(x)
>>> cos = np.cos(x)

>>> fig, ax = plt.subplots()

>>> ax.plot(x, sin)
[<matplotlib.lines.Line2D at 0x1247b6310>]

>>> ax.plot(x, cos)
[<matplotlib.lines.Line2D at 0x112b0d4c0>]
```




**Nota:** Los colores «auto» asignados a las funciones siguen un ciclo establecido por matplotlib que es igualmente personalizable.

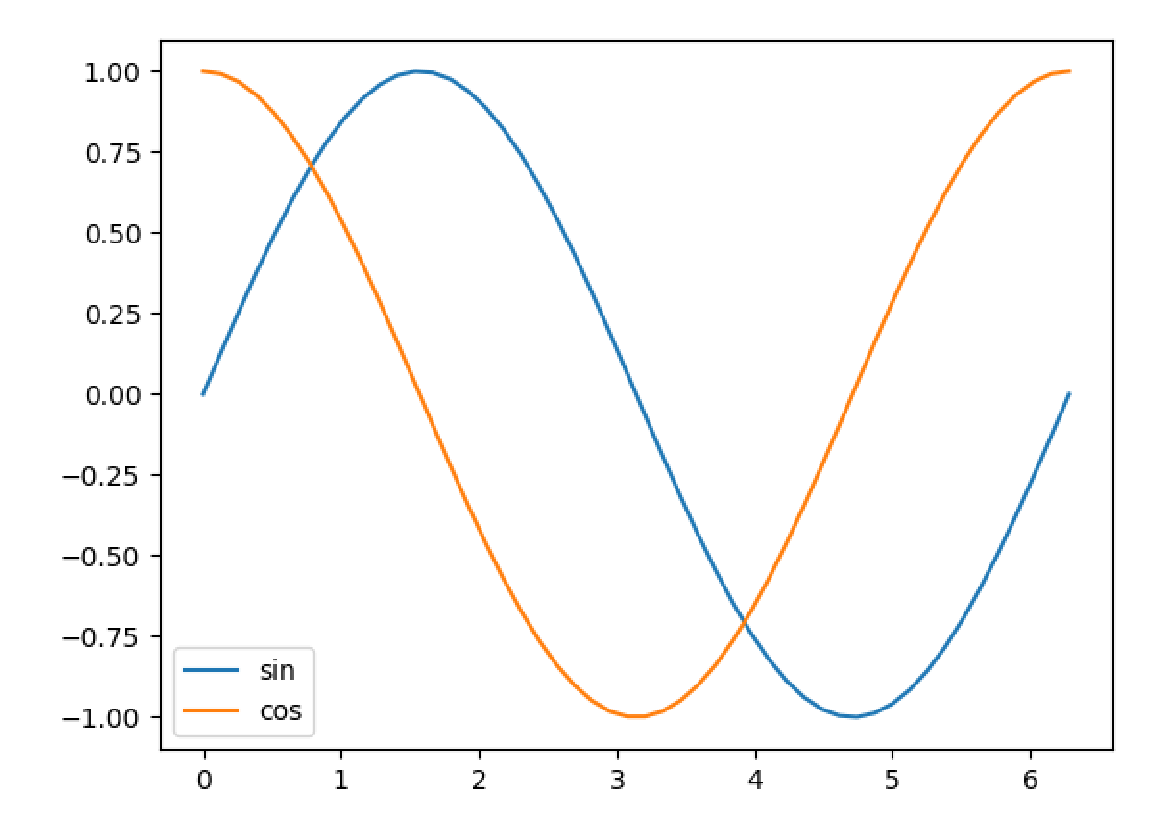
### Leyenda

En el caso de que tengamos múltiples gráficos en el mismo marco puede ser deseable mostrar una leyenda identificativa. Para usarla necesitamos asignar etiquetas a cada función. Veamos a continuación cómo incorporar una leyenda:

```
>>> ax.plot(x, sin, label='sin')
[<matplotlib.lines.Line2D at 0x124e07ac0>]

>>> ax.plot(x, cos, label='cos')
[<matplotlib.lines.Line2D at 0x123c58f10>]

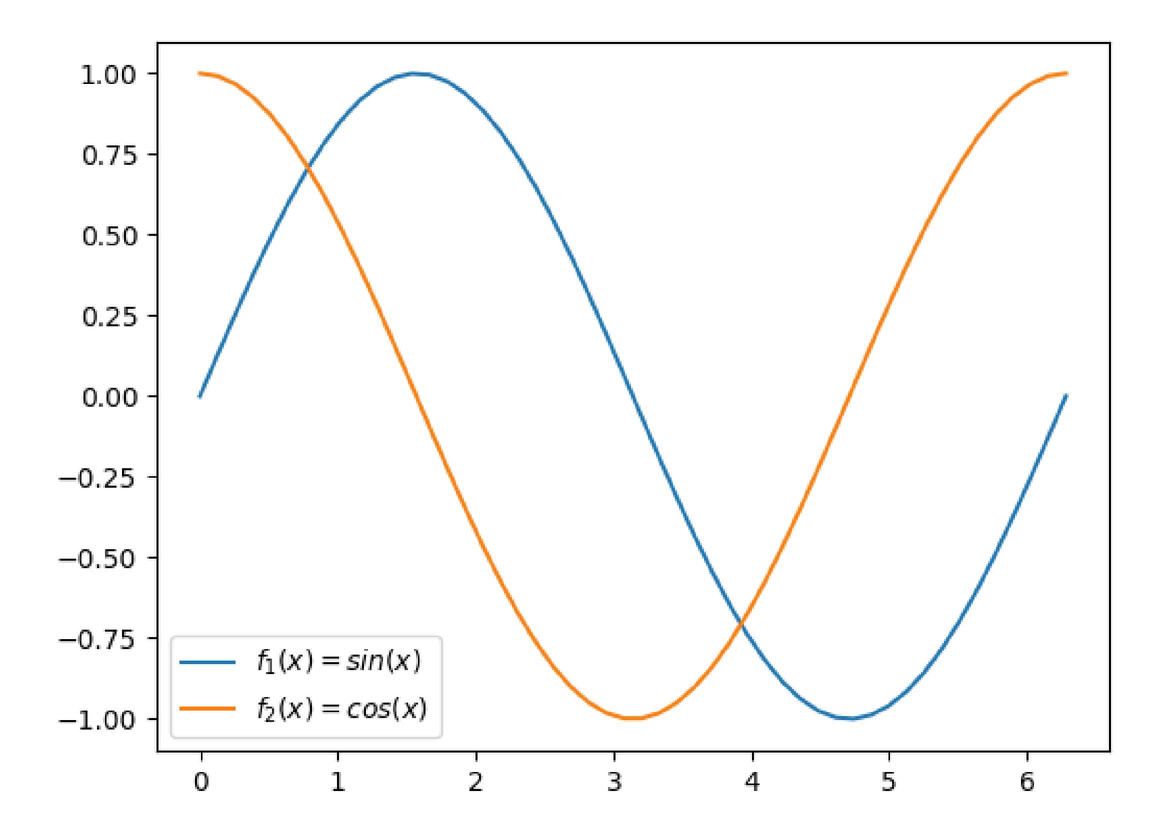
>>> ax.legend()
<matplotlib.legend.Legend at 0x123c8f190>
```




Es posible incorporar sintaxis Latex en los distintos elementos textuales de matplotlib. En el siguiente ejemplo usaremos esta notación en las etiquetas de las funciones utilizando el símbolo `$ ... $` para ello:

```
>>> ax.plot(x, sin, label='$f_1(x) = sin(x)$')
[<matplotlib.lines.Line2D at 0x11682f3a0>]

>>> ax.plot(x, cos, label='$f_2(x) = cos(x)$')
[<matplotlib.lines.Line2D at 0x11682b3a0>]
```

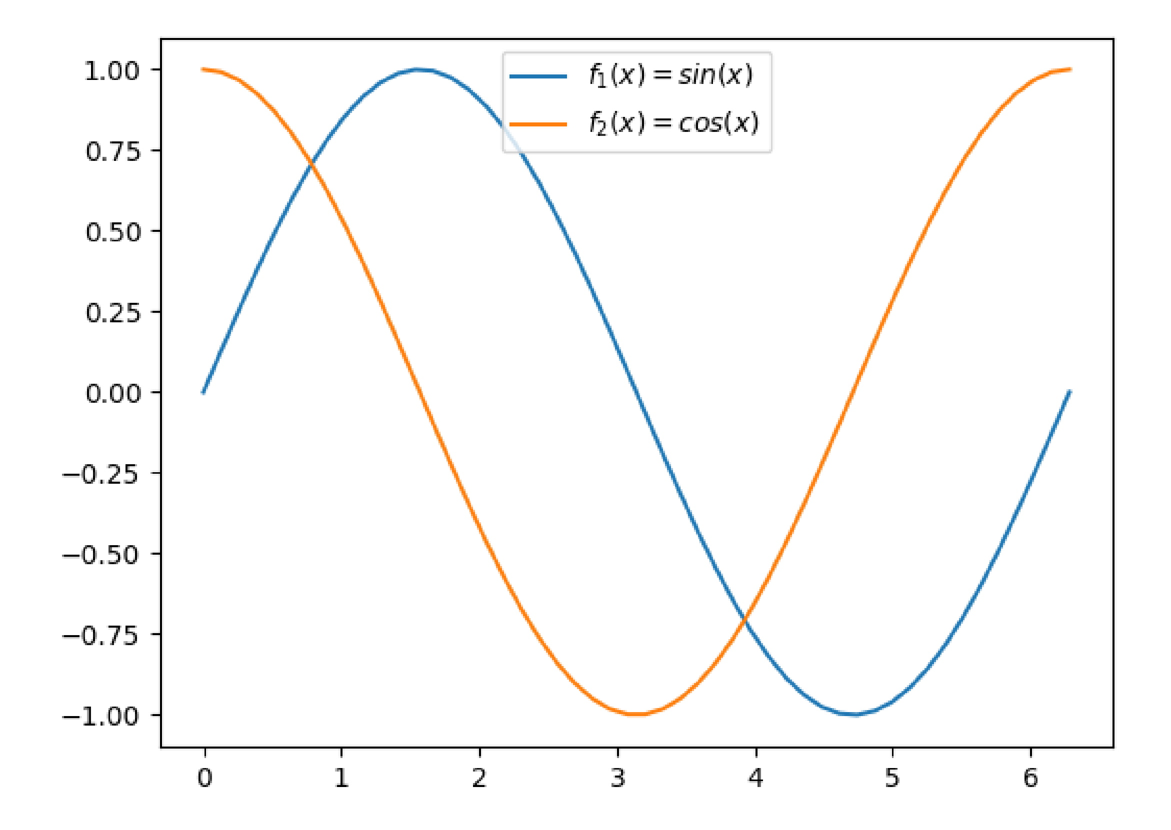



### Ubicación de la leyenda

Matplotlib intenta encontrar la **mejor ubicación** para la leyenda en el marco. Sin embargo, también es posible personalizar el lugar en el que queremos colocarla.

Si nos interesa situar la leyenda en la **parte superior central** del marco haríamos lo siguiente:

```
>>> ax.legend(loc='upper center')
<matplotlib.legend.Legend at 0x1167d43a0>
```

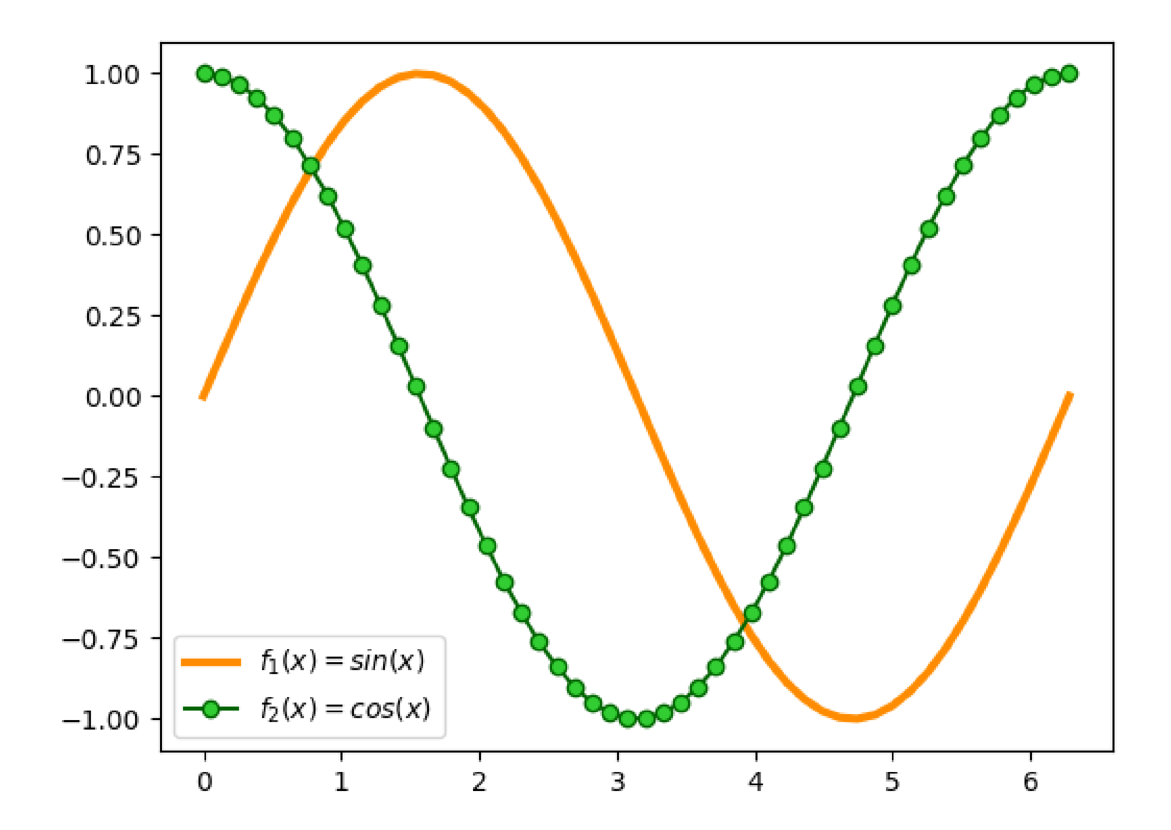



### Aplicando estilos

Para cada función que incluimos en el marco es posible establecer un estilo personalizado con multitud de parámetros. Veamos la aplicación de algunos de estos parámetros a las funciones seno y coseno con las que hemos estado trabajando:

```
>>> sin_style = dict(linewidth=3, color='darkorange')
>>> cos_style = dict(marker='o', markerfacecolor='limegreen', color='darkgreen')

>>> ax.plot(x, sin, label='$f_1(x) = sin(x)$', **sin_style)
[<matplotlib.lines.Line2D at 0x1131e9fd0>]
>>> ax.plot(x, cos, label='$f_2(x) = cos(x)$', **cos_style)
[<matplotlib.lines.Line2D at 0x1226d76d0>]
```

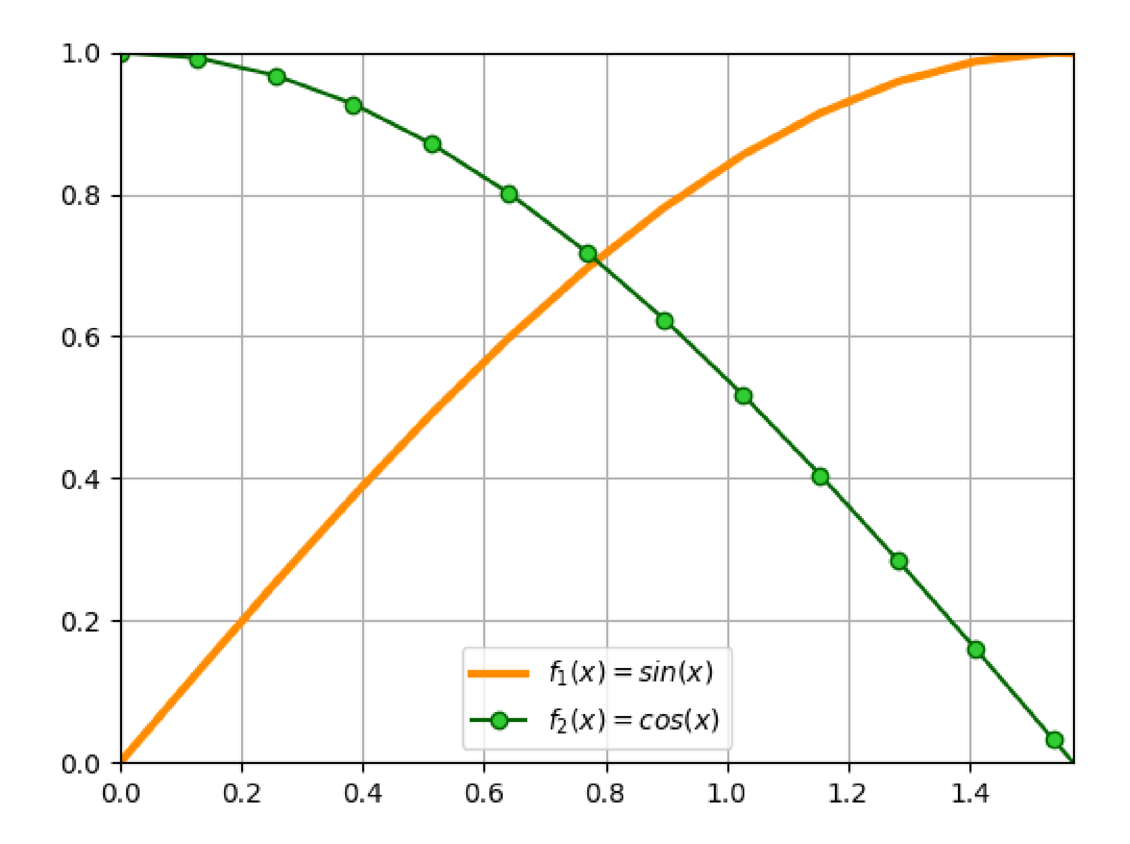



### Acotando ejes

Hay veces que nos interesa definir los límites de los ejes. En ese caso, podemos hacerlo de una manera muy sencilla:

```
>>> ax.set_xlim(0, np.pi / 2)
>>> ax.set_ylim(0, 1)

>>> ax.grid()  # sólo a efectos estéticos
```




**Truco:** También es posible especificar únicamente límite inferior o superior en ambas funciones `set_xlim()` y `set_ylim()`. En ese caso, el otro valor sería ajustado automáticamente por matplotlib.

### Anotaciones

En ocasiones necesitamos añadir ciertas anotaciones al gráfico que estamos diseñando. Esto permite destacar áreas o detalles que pueden ser relevantes.

Partiendo de las funciones seno y coseno con las que hemos estado trabajando, vamos a suponer que **queremos obtener sus puntos de corte**, es decir, resolver la siguiente ecuación:

$$sin(x) = cos(x)$$
$$\Downarrow$$
$$x = \frac{\pi}{4} + \pi n, \ n \in \mathbb{Z}$$

Para el caso que nos ocupa haríamos $n = 0$ con lo que obtendríamos la siguiente solución:

```
>>> xsol = np.pi / 4 + np.pi * 0
>>> ysol = np.sin(xsol)

>>> xsol, ysol
(0.7853981633974483, 0.7071067811865475)
```

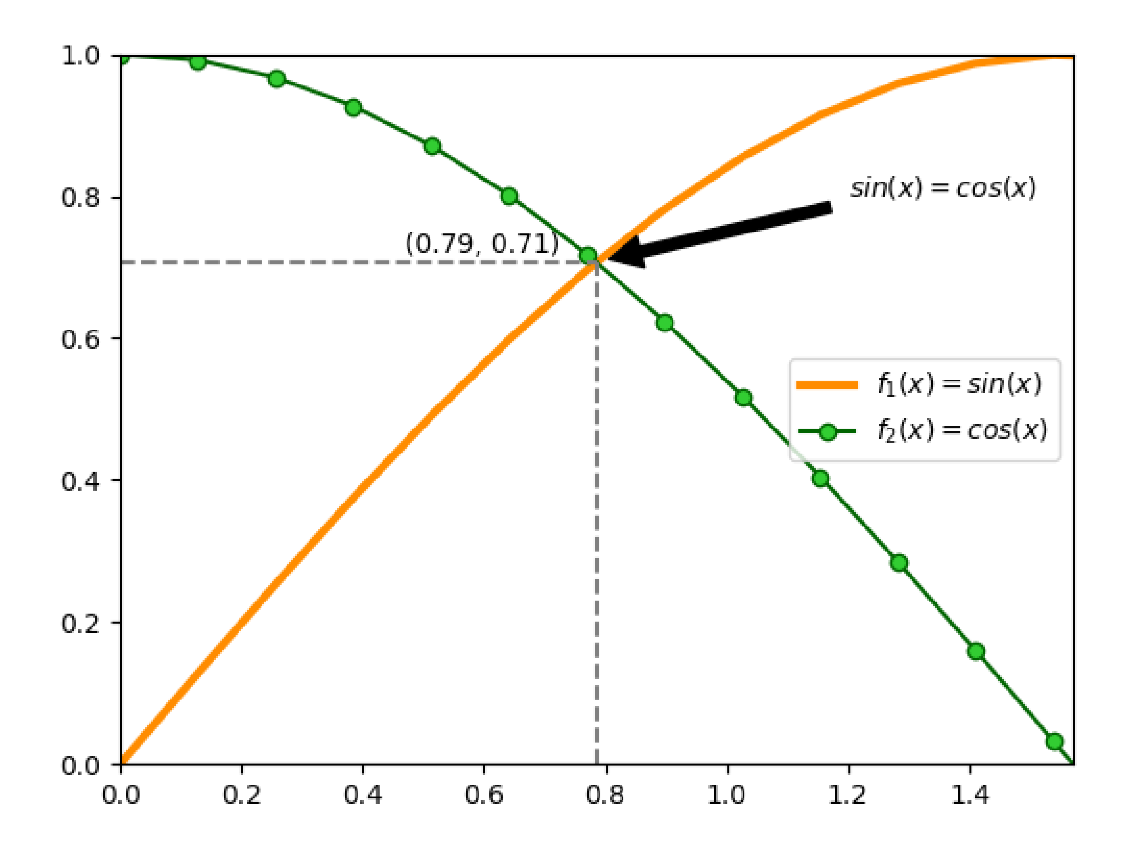
Vamos a insertar una serie de anotaciones en el gráfico:

- Flecha en el punto de corte con etiqueta de ecuación.
- Coordenadas de solución en el punto de corte.
- Proyección del punto de corte hacia ambos ejes.

```
>>> ax.annotate('$sin(x) = cos(x)$',
...             xy=(xsol, ysol),
...             xytext=(1.2, 0.8),
...             arrowprops=dict(facecolor='black', shrink=0.05))

>>> ax.text(0.47, 0.72, f'({xsol:.2f}, {ysol:.2f})')

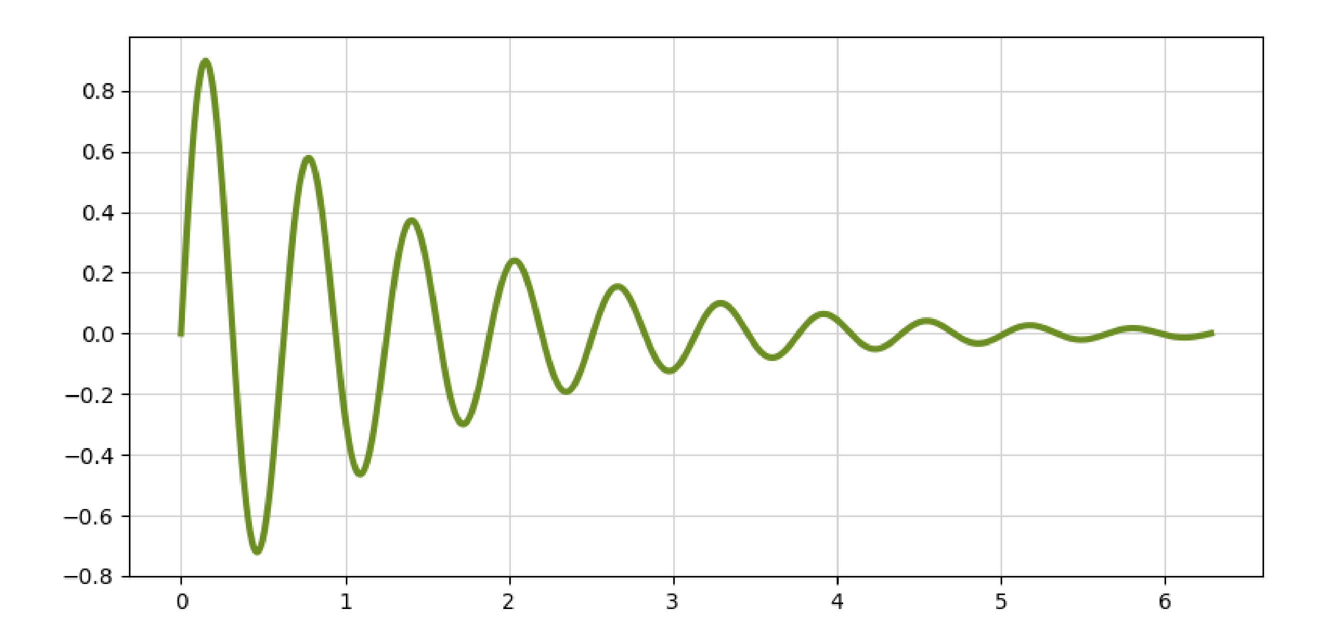
>>> ax.plot([xsol, xsol], [0, ysol], color='gray', linestyle='--')
>>> ax.plot([0, xsol], [ysol, ysol], color='gray', linestyle='--')
```




---

**Ejercicio**

Escriba el código Python necesario para obtener el siguiente gráfico:




Datos:

- $x \in [0, 2\pi]$ (1000 puntos)
- $y = e^{-\alpha x} sin(\beta x)$, donde $\alpha = 0.7$ y $\beta = 10$.

---

## 9.4.4 Tipos de gráficos

Mediante matplotlib podemos hacer prácticamente cualquier tipo de gráfico. En esta sección haremos un repaso por algunos de ellos.

### Gráficos de barras

Vamos a partir de un «dataset» que contiene los resultados de los Juegos Olímpicos de Tokio 2020. Hemos descargado el fichero `medals.xlsx` desde una página de Kaggle[4].

En primer lugar cargaremos este fichero en un DataFrame y haremos una pequeña «limpieza»:

```
>>> df = pd.read_excel('pypi/datascience/files/medals.xlsx')

>>> df.head()
Rank                    Team/NOC  Gold  Silver  Bronze  Total  Rank by Total
0     1    United States of America    39      41      33    113              1
```

(continué en la próxima página)

[4] Kaggle es un servicio web que ofrece una gran variedad de «datasets», así como código, cursos y otros recursos en relación con la ciencia de datos.

(proviene de la página anterior)

```
1     2  People's Republic of China    38      32      18      88             2
2     3                       Japan    27      14      17      58             5
3     4               Great Britain    22      21      22      65             4
4     5                         ROC    20      28      23      71             3

>>> df.rename(columns={'Team/NOC': 'Country'}, inplace=True)
>>> df.set_index('Country', inplace=True)

>>> df.head()
                            Rank  Gold  Silver  Bronze  Total  Rank by Total
Country
United States of America       1    39      41      33    113              1
People's Republic of China     2    38      32      18     88              2
Japan                          3    27      14      17     58              5
Great Britain                  4    22      21      22     65              4
ROC                            5    20      28      23     71              3
```

**Importante:** Para la carga de ficheros Excel, es necesario instalar un paquete adicional denominado `openpyxl`.

A continuación crearemos un **gráfico de barras con las medallas de oro, plata y bronce de los 10 primeros países ordenados por su ranking**. Lo primero será crear el subconjunto de datos sobre el que vamos a trabajar. Hay muchas maneras de hacerlo. Una de ellas:

```
>>> df_best = df.nsmallest(10, 'Rank')

>>> df_best
                            Rank  Gold  Silver  Bronze  Total  Rank by Total
Country
United States of America       1    39      41      33    113              1
People's Republic of China     2    38      32      18     88              2
Japan                          3    27      14      17     58              5
Great Britain                  4    22      21      22     65              4
ROC                            5    20      28      23     71              3
Australia                      6    17       7      22     46              6
Netherlands                    7    10      12      14     36              9
France                         8    10      12      11     33             10
Germany                        9    10      11      16     37              8
Italy                         10    10      10      20     40              7
```

Ahora ya podemos centrarnos en el diseño del gráfico de barras:

```
>>> fig, ax = plt.subplots(figsize=(8, 5), dpi=100)  # 800x500 px

>>> bar_width = 0.30
>>> x = np.arange(df_best.index.size)

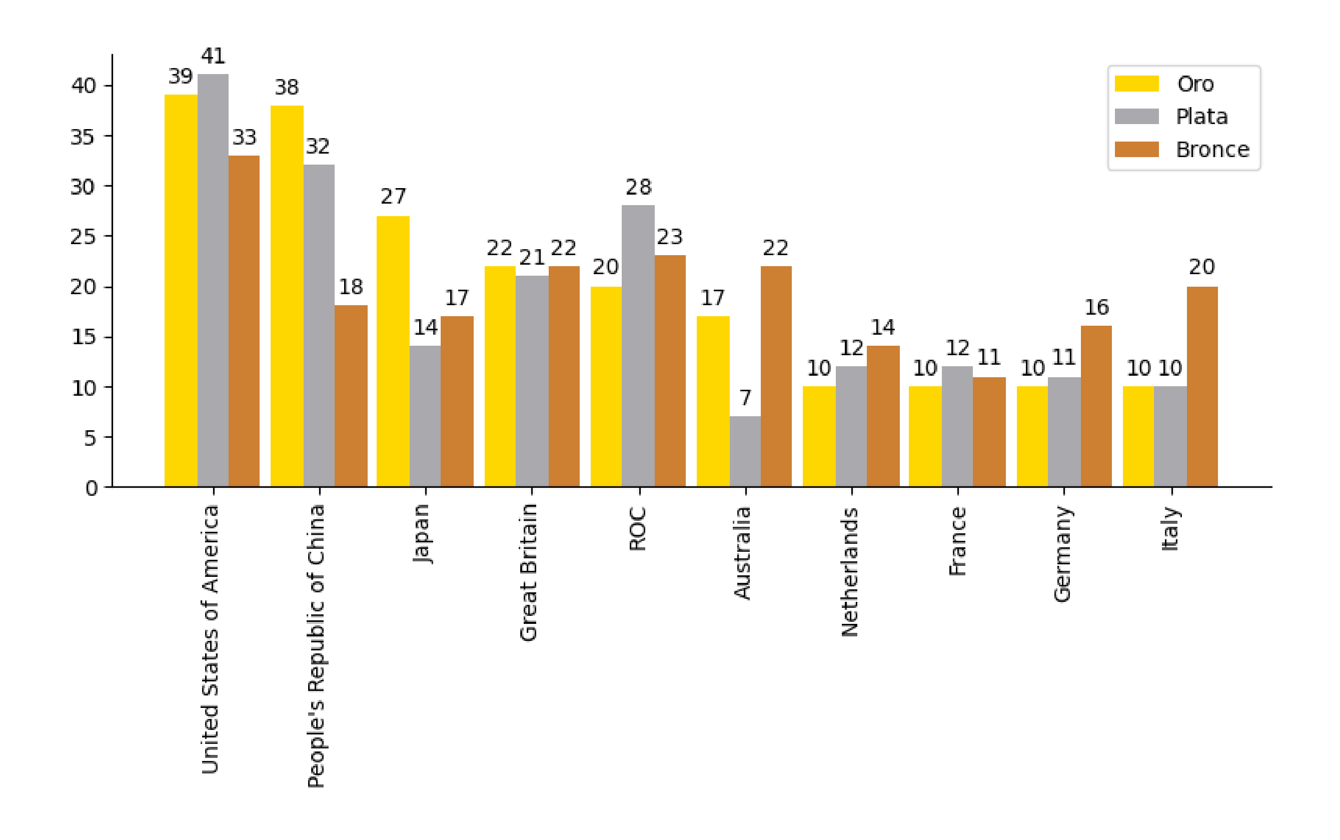
>>> golden_medals = ax.bar(x - bar_width, df_best['Gold'],
...                        bar_width, label='Oro', color='#ffd700')
>>> silver_medals = ax.bar(x, df_best['Silver'],
...                        bar_width, label='Plata', color='#aaa9ad')
>>> bronze_medals = ax.bar(x + bar_width, df_best['Bronze'],
...                        bar_width, label='Bronce', color='#cd7f32')

>>> ax.set_xticks(x)
>>> ax.set_xticklabels(df_best.index, rotation=90)
>>> ax.legend()

>>> # Etiquetas en barras
>>> ax.bar_label(golden_medals, padding=3)
>>> ax.bar_label(silver_medals, padding=3)
>>> ax.bar_label(bronze_medals, padding=3)

>>> ax.spines['right'].set_visible(False)  # ocultar borde derecho
>>> ax.spines['top'].set_visible(False)    # ocultar borde superior

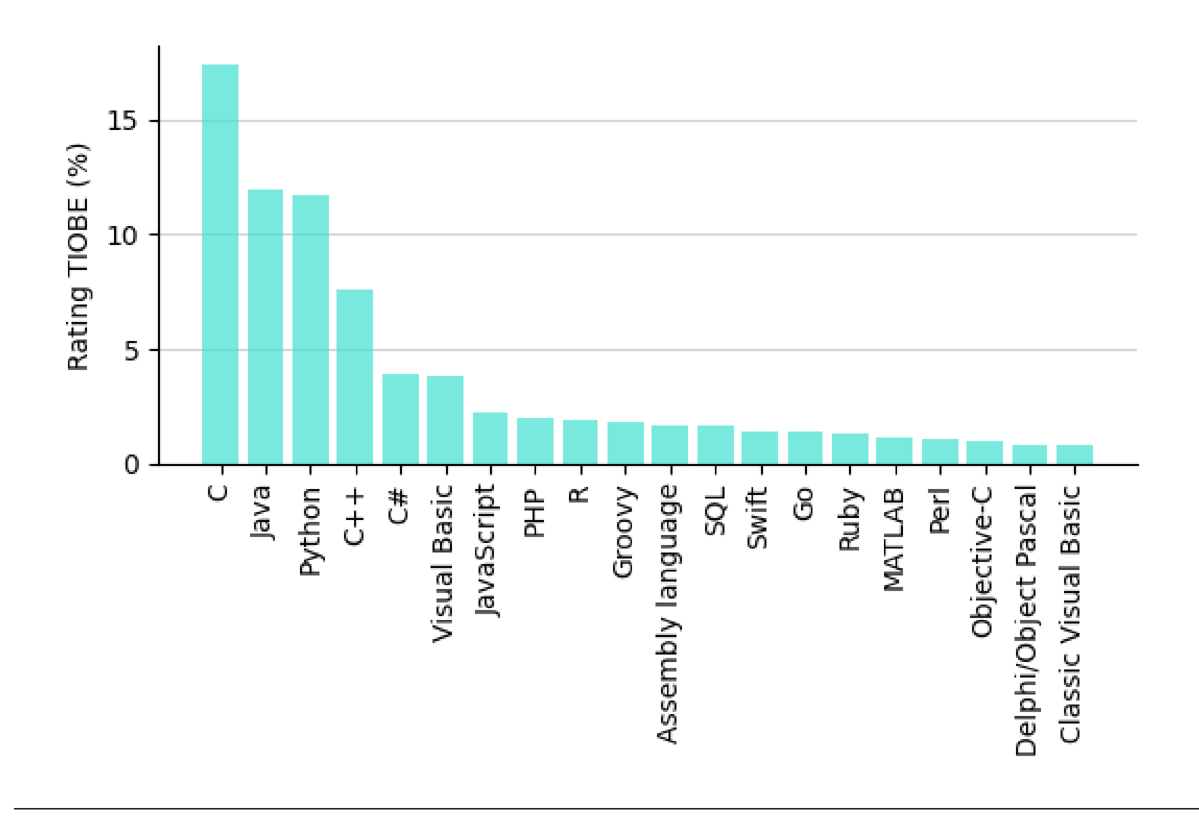
>>> fig.tight_layout()  # ajustar elementos al tamaño de la figura
>>> fig
```




### Ejercicio

Partiendo del fichero `tiobe-2020-clean.csv` que contiene las valoraciones de los lenguajes de programación más usados durante el año 2020 (según el índice TIOBE)[8], cree el siguiente gráfico de barras:

[8] Datos extraídos desde esta página de Kaggle.




### Gráficos de dispersión

Para este gráfico vamos a usar un «dataset» de jugadores de la NBA[5] extraído desde esta página de Kaggle. El fichero `nba-data.csv` contiene información desde 1996 hasta 2019.

En primer lugar cargamos los datos y nos quedamos con un subconjunto de las columnas:

```
>>> df = pd.read_csv('pypi/datascience/files/nba-data.csv', usecols=['pts', 'reb',
↪'ast'])

>>> df.head()
   pts  reb  ast
0  4.8  4.5  0.5
1  0.3  0.8  0.0
2  4.5  1.6  0.9
3  7.8  4.4  1.4
4  3.7  1.6  0.5

>>> df.shape
(11700, 3)
```

[5] National Basketball League (liga estadounidense de baloncesto).

El objetivo es crear un **gráfico de dispersión en el relacionaremos los puntos anotados con los rebotes capturados, así como las asistencias dadas**:

```
>>> fig, ax = plt.subplots(figsize=(8, 6), dpi=100)  # 800x600 px

>>> # Crear variables auxiliares
>>> x = df['pts']
>>> y = df['reb']
>>> colors = df['ast']

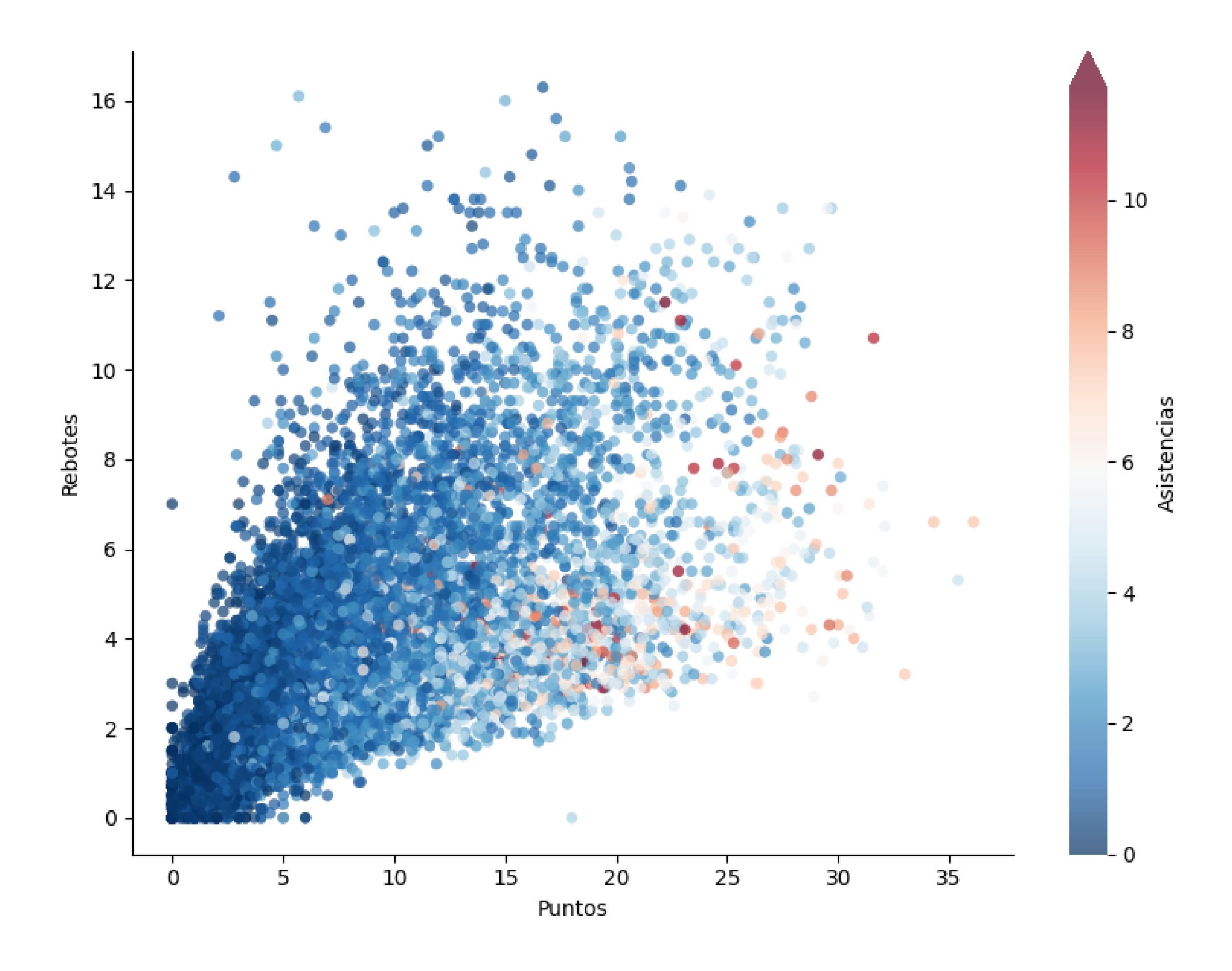
>>> p = ax.scatter(x, y,
...                s=30,  # tamaño de los puntos
...                c=colors, cmap='RdBu_r',  # colores
...                vmin=colors.min(), vmax=colors.max(),  # normalización de colores
...                alpha=0.7,
...                edgecolors='none')

>>> # Barra de colores
>>> cb = fig.colorbar(p, ax=ax, label='Asistencias', extend='max')
>>> cb.outline.set_visible(False)

>>> ax.set_xlabel('Puntos')
>>> ax.set_ylabel('Rebotes')

>>> ax.spines['right'].set_visible(False)
>>> ax.spines['top'].set_visible(False)

>>> fig.tight_layout()
```



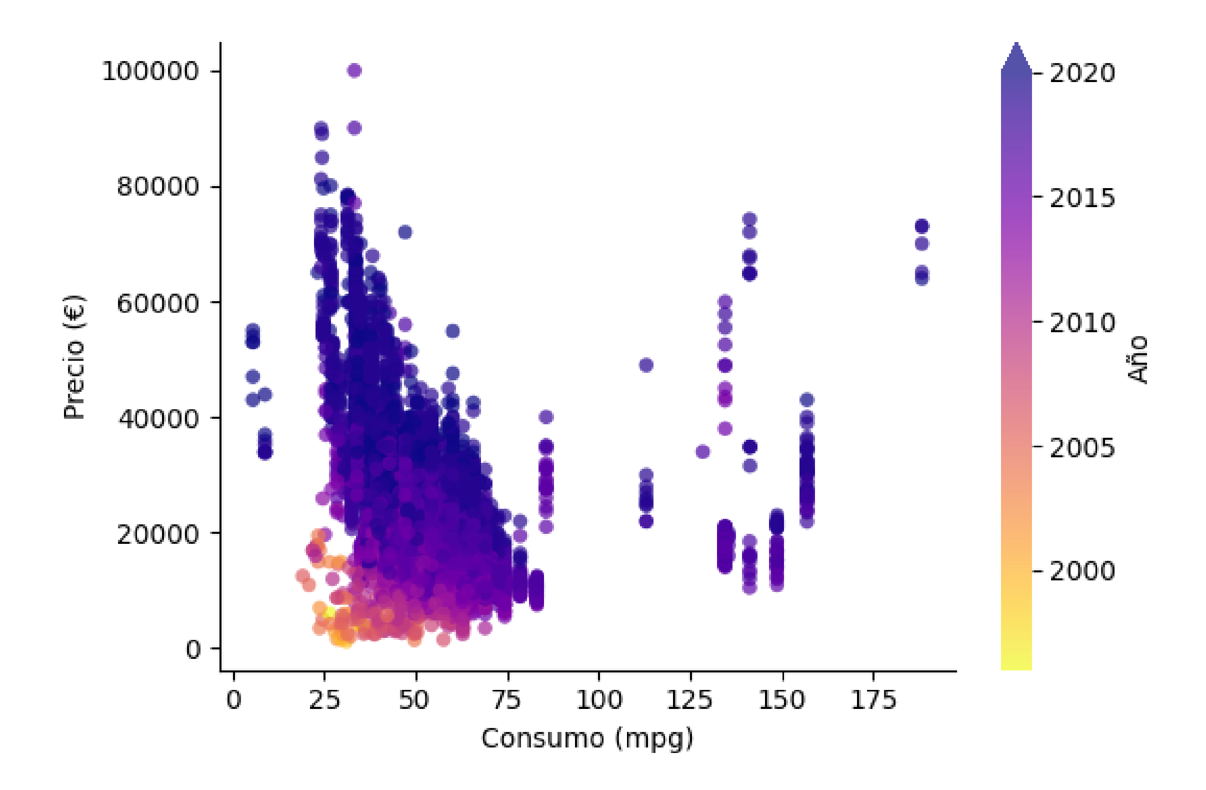

Del gráfico anterior cabe destacar varios aspectos:

- **Normalización**: Cuando aplicamos una estética de color al gráfico basada en los datos de una variable, debemos normalizar dicha variable en el mapa de color («colormap») que elijamos. Para ello, matplotlib nos ofrece la normalización de mapas de color. En el caso concreto de `scatter()` pasaríamos esta normalización mediante el parámetro `norm` pero también podemos usar los parámetros `vmin` y `vmax`.
- **Barra de color**: Se trata de una leyenda particular en la que mostramos el gradiente de color vinculado a una determinada estética/variable del gráfico. Matplotlib también nos permite personalizar estas barras de color.

**Ejercicio**

Partiendo del fichero `bmw-clean.csv` que contiene información sobre vehículos de la marca BMW[9], cree el siguiente gráfico de dispersión:

[9] Datos extraídos desde esta página de Kaggle.




El mapa de color que se ha usado es `plasma_r`.

---

### Histogramas

En esta ocasión vamos a trabajar con un «dataset» de «Avengers»[6] extraído desde Kaggle. Hemos descargado el fichero `avengers.csv`.

Como punto de partida vamos a cargar la información y a quedarnos únicamente con la columna que hace referencia al año en el que se crearon los personajes:

```
>>> df = pd.read_csv('pypi/datascience/files/avengers.csv', usecols=['Year'])

>>> df.head()
Year
0  1963
1  1963
2  1963
3  1963
4  1963

>>> df.shape
(173, 1)
```

[6] Los Vengadores son un equipo de superhéroes publicados por Marvel Comics.

Igualmente haremos un pequeño filtrado para manejar sólo registros a partir de 1960:

```
>>> df = df[df['Year'] >= 1960]

>>> df.shape
(159, 1)
```

Ahora ya podemos construir el histograma, que va a representar las **frecuencias absolutas de creación de personajes Marvel según su año de creación**.

Aunque es posible indicar un número determinado de contenedores («bins»), en este caso vamos a especificar directamente los intervalos (cada 5 años):

```
>>> df['Year'].min(), df['Year'].max()
(1963, 2015)

>>> bins = range(1960, 2021, 5)
```

Y a continuación el código necesario para montar el gráfico:

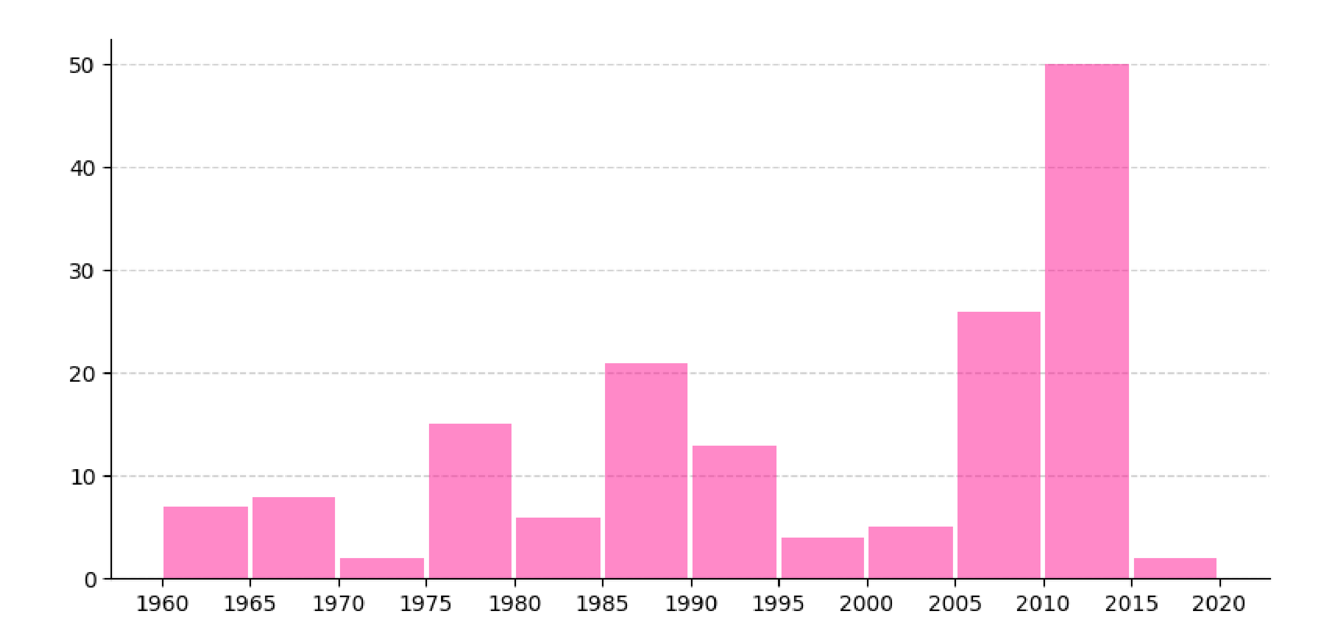
```
>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100)  # 800x400 px

>>> ax.hist(df,
...         bins=bins,     # intervalos de agrupación
...         rwidth=0.95,   # ancho de cada barra
...         zorder=2,      # barras por encima de rejilla
...         color='deeppink',
...         alpha=0.5)

>>> ax.spines['right'].set_visible(False)
>>> ax.spines['top'].set_visible(False)

>>> ax.set_xticks(bins)  # etiquetas de intervalos en el eje x
>>> ax.yaxis.grid(color='lightgray', linestyle='--')  # rejilla

>>> fig.tight_layout()
```



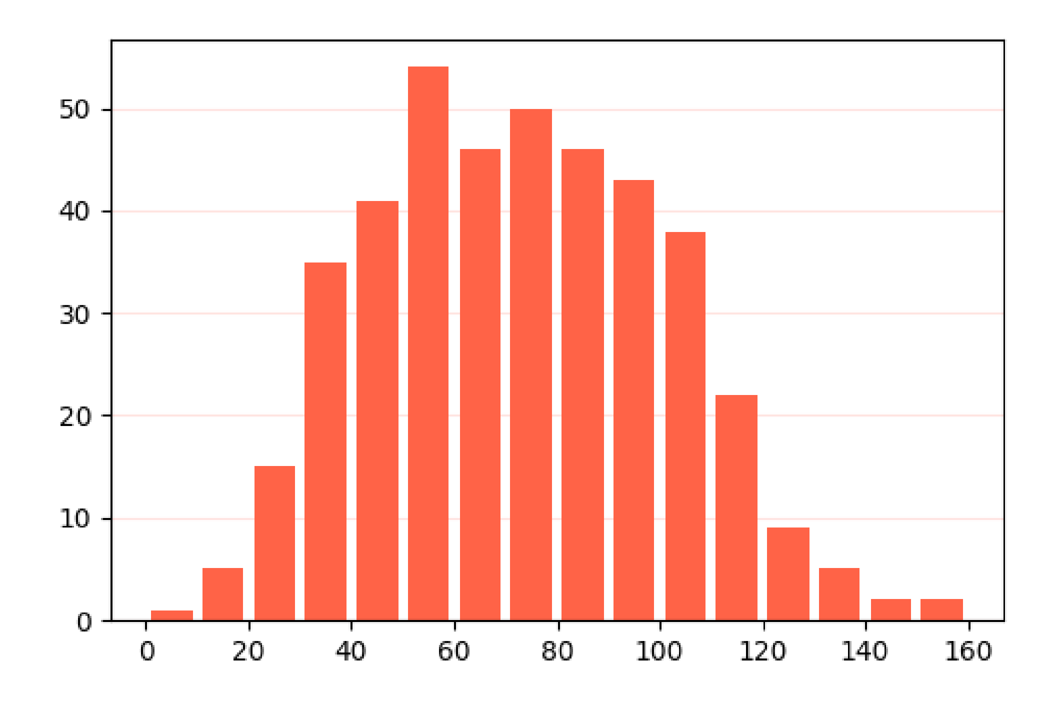

Descargo de responsabilidad: Técnicamente este gráfico no es un histograma ya que los años (fechas en general) no representan categorías válidas, pero sirve a efectos demostrativos de cómo se construyen este tipo de diagramas.

---

**Ejercicio**

Partiendo del fichero `pokemon.csv` que contiene información sobre Pokemon[10], cree el siguiente histograma en el que se analiza el número de personajes «pokemons» en función de su velocidad (columna *Speed*):

[10] Datos extraídos desde esta página de Kaggle.




### Gráficos para series temporales

Vamos a trabajar con un conjunto de datos extraído desde esta página de Kaggle que contiene información histórica de temperaturas del planeta Tierra. El fichero `global-temperatures.csv` se ha descargado para su tratamiento.

En primer lugar cargamos los datos, renombramos las columnas y eliminamos los valores nulos:

```
>>> df = pd.read_csv('pypi/datascience/files/global-temperatures.csv',
...                  parse_dates=['dt'],  # conversión a tipo datetime
...                  usecols=['dt', 'LandAverageTemperature'])

>>> df.rename(columns={'dt': 'when', 'LandAverageTemperature': 'temp'}, inplace=True)
>>> df.dropna(inplace=True)

>>> df.head()
        when    temp
0 1750-01-01   3.034
1 1750-02-01   3.083
2 1750-03-01   5.626
3 1750-04-01   8.490
4 1750-05-01  11.573
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> df.shape
(3180, 2)
```

A continuación montamos un gráfico en el que se representan todas las **mediciones históricas de la temperatura media global del planeta** y añadimos una línea de tendencia:

```
>>> # Necesitamos algunas utilidades de gestión de fechas
>>> from matplotlib.dates import YearLocator, DateFormatter, date2num
>>> from matplotlib.ticker import MultipleLocator

>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100)  # 800x400 px

>>> # Alias para simplificar el acceso
>>> x = df.when
>>> y = df.temp

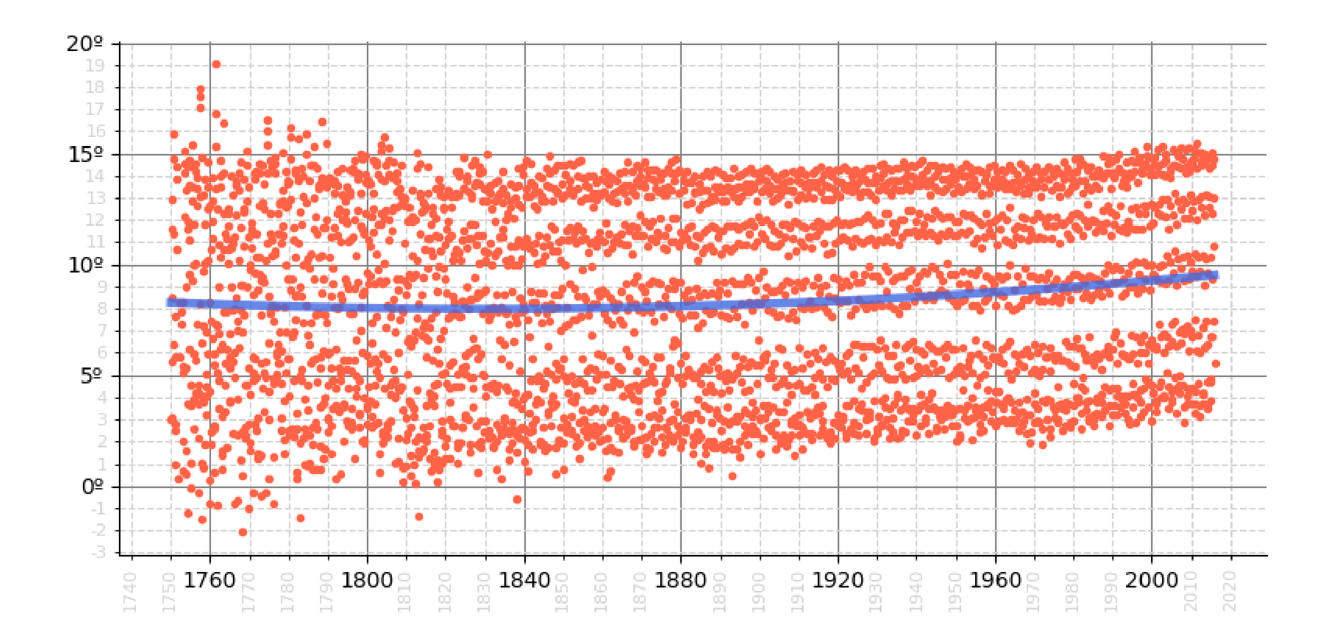
>>> ax.plot(x, y,
...         linestyle='None', marker='.', color='tomato',  # estilo de línea
...         zorder=2)  # orden para colocar sobre rejilla

>>> # Construcción de la línea de tendencia
>>> x = date2num(x)
>>> z = np.polyfit(x, y, 2)  # ajuste polinómico de grado 2
>>> p = np.poly1d(z)
>>> plt.plot(x, p(x), linewidth=4, alpha=0.8, color='royalblue')

>>> # Formateo de los ejes
>>> ax.xaxis.set_minor_locator(YearLocator(10))
>>> ax.xaxis.set_minor_formatter(DateFormatter('%Y'))
>>> ax.tick_params(axis='x', which='minor',
...                labelsize=8, labelcolor='lightgray', rotation=90)
>>> ax.xaxis.grid(which='minor', color='lightgray', linestyle='dashed')
>>> ax.yaxis.set_major_formatter('{x:.0f}º')
>>> ax.yaxis.set_minor_locator(MultipleLocator(1))
>>> ax.tick_params(axis='y', which='minor',
...                labelsize=8, labelcolor='lightgray')
>>> ax.yaxis.grid(which='minor', linestyle='dashed', color='lightgray')
>>> ax.yaxis.set_minor_formatter('{x:.0f}')
>>> ax.tick_params(axis='y', which='minor', labelsize=8, labelcolor='lightgray')

>>> ax.spines['right'].set_visible(False)
>>> ax.spines['top'].set_visible(False)

>>> fig.tight_layout()
```




### Mapas de calor

Para este tipo de gráfico vamos a utilizar un «dataset» que recoge las 1000 películas más valoradas en IMDB[7]. Está sacado desde esta página de Kaggle y se ha descargado el fichero de datos en `imdb-top-1000.csv`.

En primer lugar vamos a cargar los datos quedándonos con las columnas *Certificate* (clasificación de la película según edades), *Genre* (géneros de la película) e *IMDB_Rating* (valoración de la película en IMDB):

```
>>> df = pd.read_csv('pypi/datascience/files/imdb-top-1000.csv',
...                  usecols=['Certificate', 'Genre', 'IMDB_Rating'])

>>> df.head()
  Certificate                  Genre  IMDB_Rating
0           A                  Drama          9.3
1           A           Crime, Drama          9.2
2          UA   Action, Crime, Drama          9.0
3           A           Crime, Drama          9.0
4           U           Crime, Drama          9.0
```

Ahora creamos una nueva columna en el DataFrame donde guardaremos únicamente el género principal de cada película:

```
>>> df['Main_Genre'] = df['Genre'].str.split(',', expand=True)[0]
```

(continué en la próxima página)

[7] IMDB es una reconocida página web que contiene valoraciones sobre películas y series.

(proviene de la página anterior)

```
>>> df.head()
  Certificate                  Genre  IMDB_Rating Main_Genre
0           A                  Drama          9.3      Drama
1           A           Crime, Drama          9.2      Crime
2          UA  Action, Crime, Drama          9.0     Action
3           A           Crime, Drama          9.0      Crime
4           U           Crime, Drama          9.0      Crime
```

A continuación agrupamos y obtenemos los valores medios de las valoraciones:

```
>>> # unstack permite disponer la agrupación en forma tabular (para el heatmap)
>>> ratings = df.groupby(['Certificate', 'Main_Genre'])['IMDB_Rating'].mean().
↪unstack()

>>> # Nos quedamos con un subconjunto de certificados y géneros
>>> review_certificates = ['U', 'UA', 'PG-13', 'R', 'A']
>>> review_genres = ['Animation', 'Action', 'Adventure', 'Biography',
...                  'Comedy', 'Crime', 'Drama']
>>> ratings = ratings.loc[review_certificates, review_genres]

>>> # Recodificamos los certificados (clasificación) con códigos más entendibles
>>> certs_description = {'U': 'ALL', 'UA': '>12', 'PG-13': '>13', 'R': '>17', 'A': '>
↪18'}
>>> ratings.index = ratings.reset_index()['Certificate'].replace(certs_description)

>>> ratings
Main_Genre   Animation    Action  Adventure  Biography    Comedy     Crime     Drama
Certificate
ALL           7.947368  8.165000   7.953571   7.862500  7.940541  8.200000  7.976364
>12           7.883333  7.992424   7.958333   7.971429  7.885714  7.900000  7.953659
>13           7.866667  7.783333   7.600000   7.862500  7.785714  8.000000  7.775000
>17           7.800000  7.812500   7.900000   7.900000  7.824138  7.814286  7.915094
>18           7.866667  7.873171   7.912500   8.017647  7.877778  8.130233  8.036364
```

Ahora ya podemos construir el mapa de calor usando el DataFrame `ratings` generado previamente:

```
>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100)

>>> text_colors = ('black', 'white')
>>> im = ax.imshow(ratings, cmap='Reds')  # mapa de calor
>>> cbar = fig.colorbar(im, ax=ax, label='IMDB Rating')  # leyenda
>>> cbar.outline.set_visible(False)

>>> x = ratings.columns
```

(continué en la próxima página)

(proviene de la página anterior)

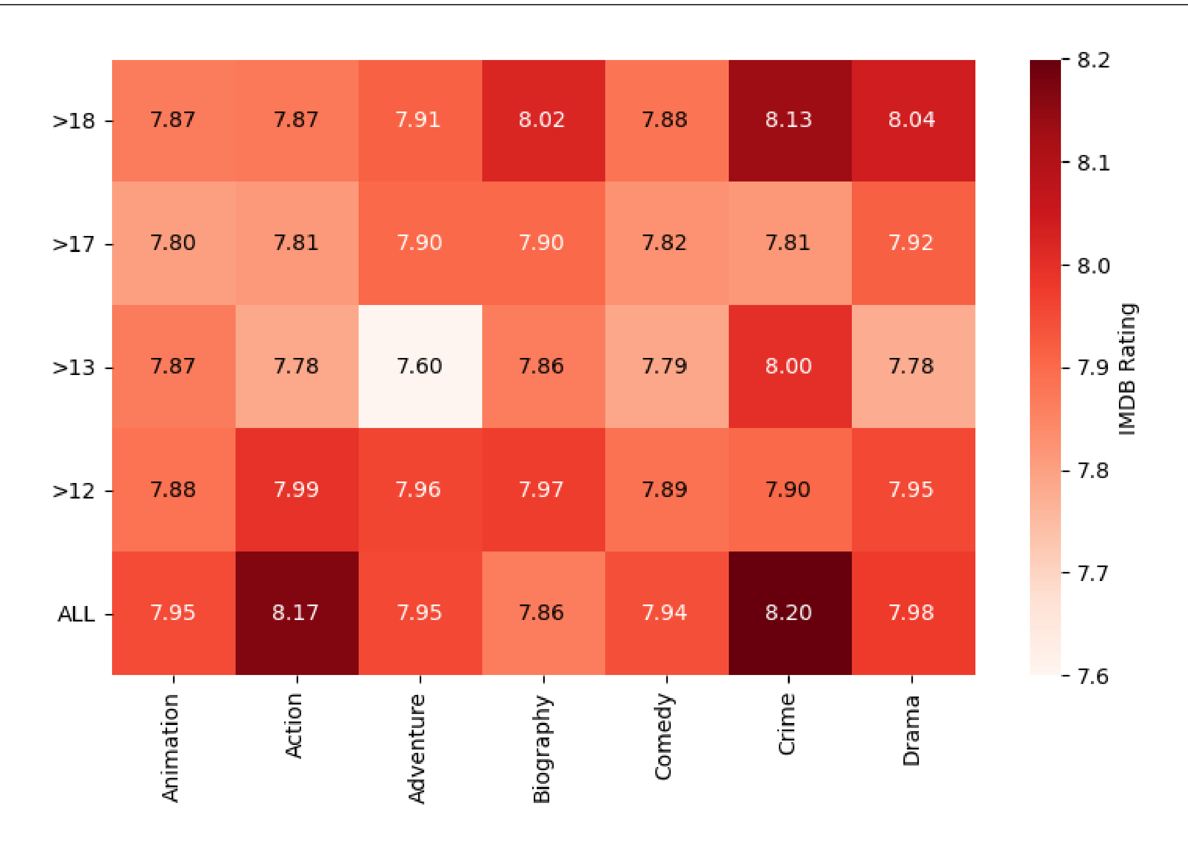
```
>>> y = ratings.index

>>> # Mostrar las etiquetas. El color del texto cambia en función de su normalización
>>> for i in range(len(y)):
...     for j in range(len(x)):
...         value = ratings.iloc[i, j]
...         text_color = text_colors[int(im.norm(value) > 0.5)]  # color etiqueta
...         ax.text(j, i, f'{value:.2f}', color=text_color, va='center', ha='center')

>>> # Formateo de los ejes
>>> ax.set_xticks(range(len(x)))
>>> ax.set_xticklabels(x, rotation=90)
>>> ax.set_yticks(range(len(y)))
>>> ax.set_yticklabels(y)
>>> ax.invert_yaxis()

>>> ax.spines[:].set_visible(False)

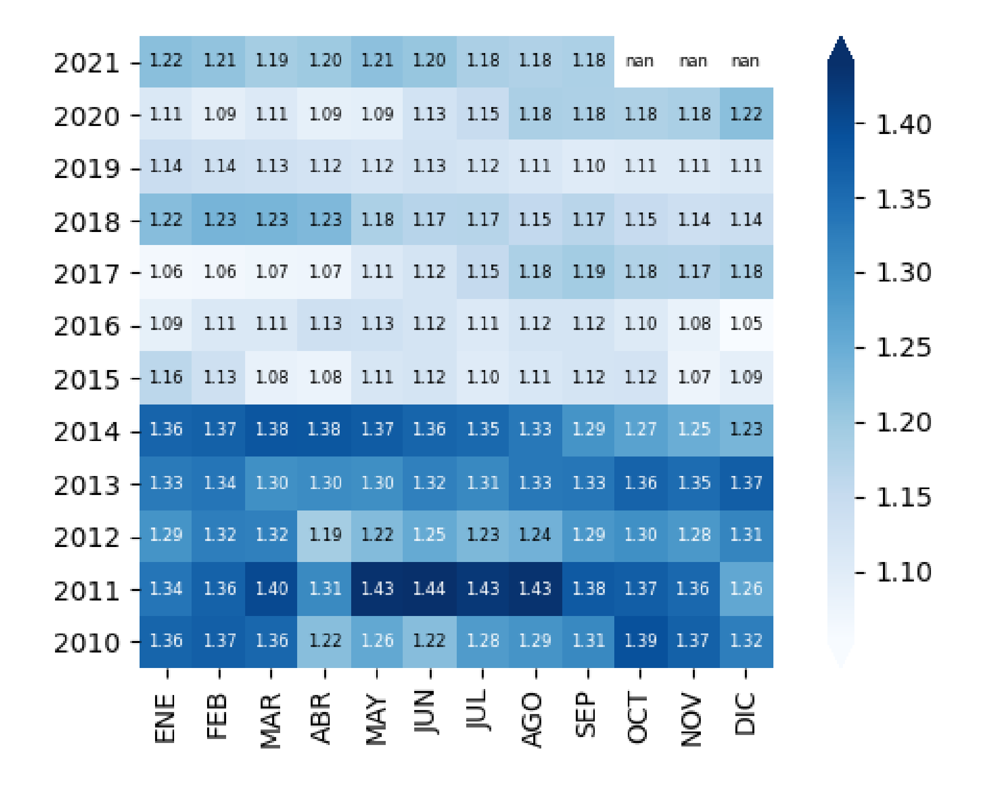
>>> fig.tight_layout()
```




### Ejercicio

Partiendo del fichero `euro-dollar-clean.csv` que contiene información sobre el cambio

euro-dollar durante los últimos 12 años[11], cree el siguiente mapa de calor en el que se analiza la evolución del cambio enfrentando meses y años:




---

## Diagramas de caja

Un diagrama de caja permite visualizar la distribución de los valores de manera rápida y muy visual:

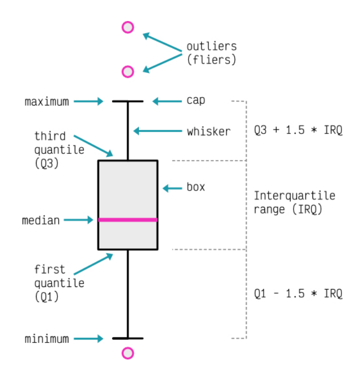



Figura 22: Anatomía de un diagrama de caja[14]

[11] Datos extraídos desde esta página de Kaggle.

[14] Inspirado en este artículo de Towards Data Science.

Para mostrar el funcionamiento de los diagramas de caja en Matplotlib vamos a hacer uso de distintas distribuciones aleatorias que crearemos mediante funciones de Numpy:

```
>>> DIST_SIZE = 100  # tamaño de la muestra

>>> boxplots = []

>>> boxplots.append(dict(
...     dist=np.random.normal(0, 1, size=DIST_SIZE),
...     label='Normal\n$\mu=0, \sigma=1$',
...     fill_color='pink',
...     brush_color='deeppink'))

>>> boxplots.append(dict(
...     dist=np.random.geometric(0.4, size=DIST_SIZE),
...     label='Geometric\n$p=0.4$',
...     fill_color='lightblue',
...     brush_color='navy'))

>>> boxplots.append(dict(
...     dist=np.random.chisquare(2, size=DIST_SIZE),
...     label='Chi-squared\n$df=2$',
...     fill_color='lightgreen',
...     brush_color='darkgreen'))
```

Ahora ya podemos construir el gráfico de cajas que nos permite visualizar la distribución de las muestras:

```
>>> fig, ax = plt.subplots(figsize=(8, 6), dpi=100)  # 800x600 px

>>> for i, boxplot in enumerate(boxplots):
...     fcolor, bcolor = boxplot['fill_color'], boxplot['brush_color']
...     ax.boxplot(boxplot['dist'],
...                labels=[boxplot['label']],
...                positions=[i],
...                widths=[.3],
...                notch=True,
...                patch_artist=True,
...                boxprops=dict(edgecolor=bcolor,
...                              facecolor=fcolor,
...                              linewidth=2),
...                capprops=dict(color=bcolor, linewidth=2),
...                flierprops=dict(color=bcolor,
...                                markerfacecolor=fcolor,
...                                linestyle='none',
...                                markeredgecolor='none',
...                                markersize=9),
```

(continué en la próxima página)

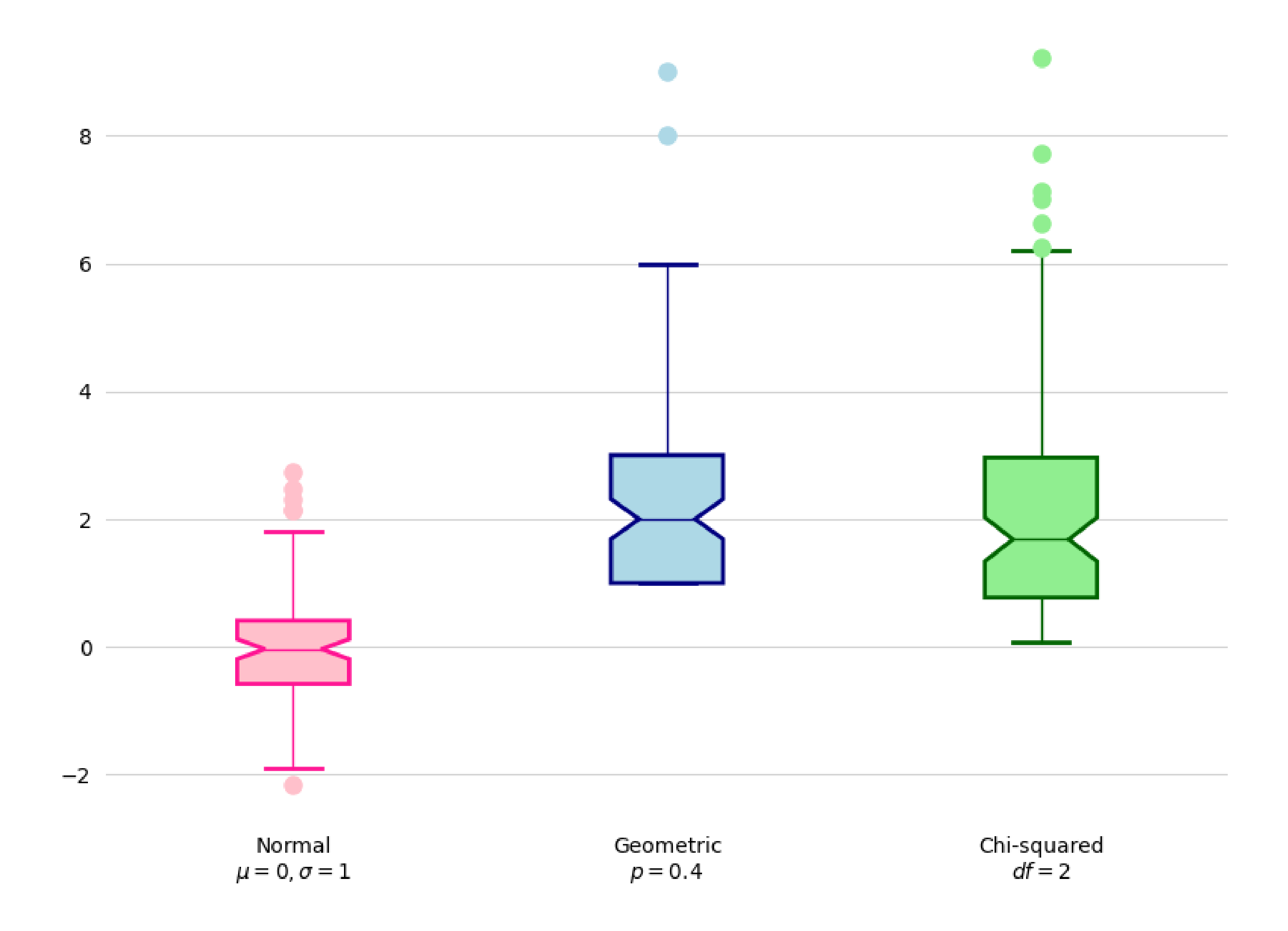
(proviene de la página anterior)

```
...                   medianprops=dict(color=bcolor),
...                   whiskerprops=dict(color=bcolor,
...                                     linewidth=1))

>>> ax.yaxis.grid(color='lightgray')
>>> ax.xaxis.set_ticks_position('none')
>>> ax.yaxis.set_ticks_position('none')

>>> ax.spines[:].set_visible(False)

>>> fig.tight_layout()
```




**Consejo:** El código para preparar el gráfico se ha complicado porque se ha incidido en mejorar la estética. En cualquier caso, una vez hecho, se puede refactorizar en una función y reutilizarlo para futuros trabajos.

### Gráficos de evolución

Partiendo de un conjunto de datos temporales, vamos a aprovechar para elaborar un gráfico de evolución del precio de criptomonedas. En esta ocasión hemos utilizado el «dataset» `eth-usd.csv` descargado desde esta página de Kaggle. Contiene la valoración de la criptomoneda **Ethereum** en función de una marca temporal, así como el volumen de «moneda» existente en cada momento.

El objetivo será crear un **gráfico que represente el valor de la criptomoneda (a lo largo del tiempo) en contraposición al volumen de unidades**.

Lo primero que haremos, además de cargar los datos, será lo siguiente:

- Seleccionar las columnas *Date* (fecha de referencia), *Open* (precio de la moneda a la apertura) y *Volume* (volumen de moneda).
- Parsear el campo fecha.
- Filtrar sólo aquellos registros a partir del 1 de enero de 2017 (por simplicidad).
- Dividir la columna de volumen por 10M de cara a equiparar cantidades con la valoración (ajuste de gráfico).
- Aplicar una media móvil para suavizar las curvas a representar.

```
>>> import datetime

>>> df = pd.read_csv('pypi/datascience/files/eth-usd.csv',
...                  parse_dates=['Date'],
...                  usecols=['Date', 'Open', 'Volume'],
...                  index_col='Date')

>>> min_date = datetime.datetime(year=2017, month=1, day=1)
>>> df = df.loc[df.index > min_date]

>>> df['Volume'] /= 1e7

>>> df_smooth = df.rolling(20).mean().dropna()

>>> df_smooth.head()
                 Open    Volume
Date
2017-01-21   9.968611  2.146882
2017-01-22  10.105573  2.117377
2017-01-23  10.222339  1.985587
2017-01-24  10.273270  1.821968
2017-01-25  10.239854  1.647938
```

Ahora ya podemos montar el gráfico dedicando algo de esfuerzo a la parte estética:

```
>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100)  # 800x400px

>>> # Alias para facilitar el acceso
>>> x = df_smooth.index
>>> y_open = df_smooth['Open']
>>> y_vol = df_smooth['Volume']

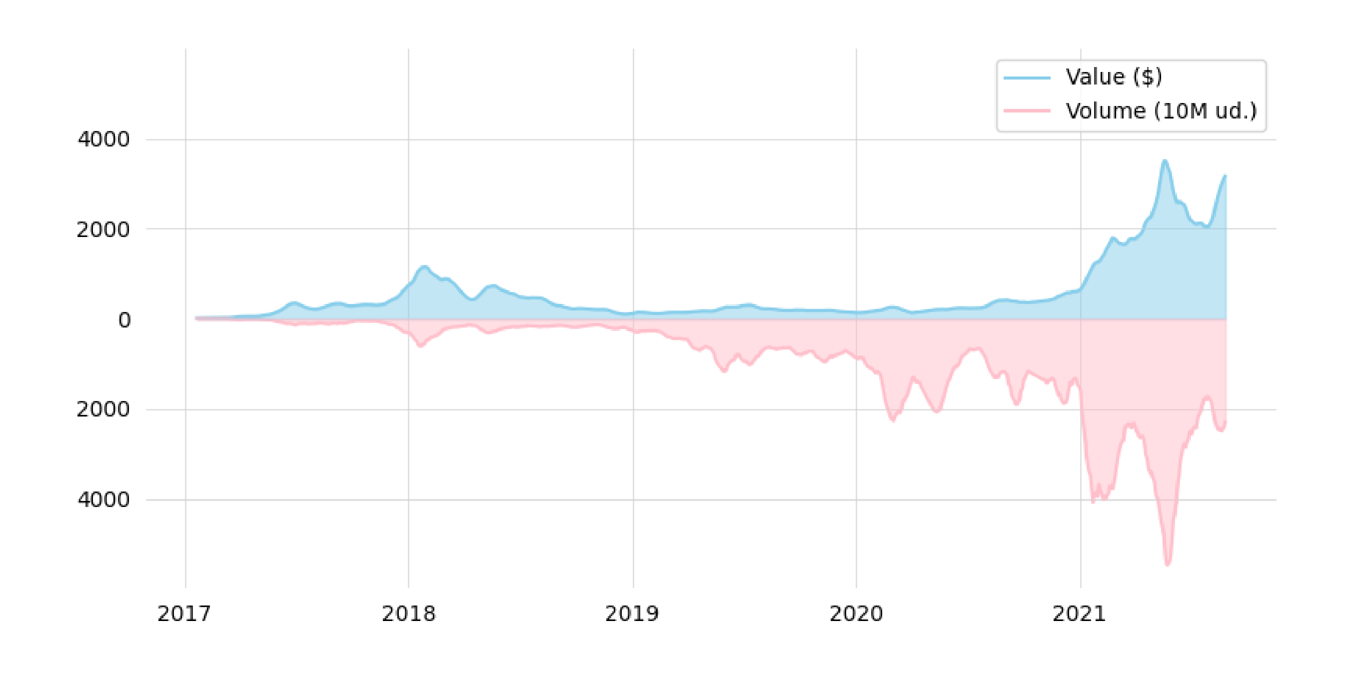
>>> # Líneas de evolución
>>> ax.plot(x, y_open, label='Value ($)', color='skyblue', linewidth=1.5)
>>> ax.plot(x, -y_vol, label='Volume (10M ud.)', color='pink', linewidth=1.5)
>>> # Relleno del área
>>> plt.fill_between(x, y_open, alpha=0.5, color='skyblue', zorder=3)
>>> plt.fill_between(x, -y_vol, alpha=0.5, color='pink', zorder=3)

>>> # Formateo de los ejes
>>> ax.xaxis.set_ticks_position('none')
>>> ax.yaxis.set_ticks_position('none')
>>> y_ticks = [-4000, -2000, 0, 2000, 4000]
>>> y_tick_labels = ['4000', '2000', '0', '2000', '4000']
>>> ax.set_yticks(y_ticks)
>>> ax.set_yticklabels(y_tick_labels)
>>> ax.set_ylim(-6000, 6000)

>>> # Rejilla
>>> ax.xaxis.grid(color='lightgray', linewidth=.5)
>>> for y_tick in y_ticks:
...     if y_tick != 0:
...         ax.axhline(y_tick, color='lightgray', linewidth=.5)

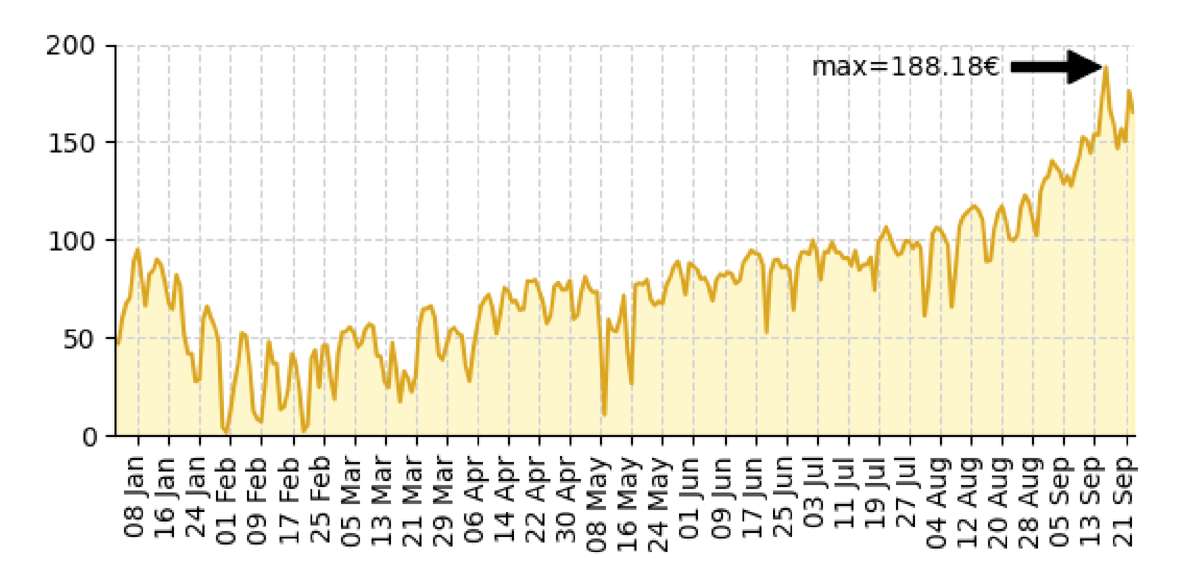
>>> ax.legend()
>>> ax.spines[:].set_visible(False)

>>> fig.tight_layout()
```




**Ejercicio**

Partiendo del fichero `mwh-spain-2021-clean.csv` que contiene información sobre el precio de la energía en España durante el año 2021[12], cree el siguiente diagrama de evolución que representa la variación del precio del MWh[13] en función del tiempo:




Las marcas (en el eje x) tienen una separación de 10 días.

[12] Datos extraídos desde esta página de El País.

[13] Mega Watio Hora (medida de consumo de energía)

CAPÍTULO 10

# Scraping

Si bien existen multitud de datos estructurados en forma de ficheros, hay otros muchos que están embebidos en páginas web y que están preparados para ser visualizados mediante un navegador.

Sin embargo, las técnicas de «scraping» nos permiten **extraer este tipo de información web** para convertirla en datos estructurados con los que poder trabajar de forma más cómoda.

Los paquetes que veremos en este capítulo bien podrían estar incluidos en otras temáticas, ya que no sólo se utilizan para «scraping».

## 10.1 requests



El paquete requests es uno de los paquetes más famosos del ecosistema Python. Como dice su lema «HTTP for Humans» permite realizar peticiones HTTP de una forma muy sencilla y realmente potente.[1]

```
$ pip install requests
```

### 10.1.1 Realizar una petición

Realizar una petición HTTP mediante *requests* es realmente sencillo:

```
>>> import requests

>>> response = requests.get('https://pypi.org')
```

Hemos ejecutado una *solicitud GET* al sitio web https://pypi.org. La respuesta se almacena en un objeto de tipo `requests.models.Response` muy rica en métodos y atributos que veremos a continuación:

```
>>> type(response)
requests.models.Response
```

[1] Foto original de portada por Frame Harirak en Unsplash.

Quizás lo primero que nos interese sea ver el contenido de la respuesta. En este sentido *requests* nos provee del atributo `text` que contendrá el **contenido html** del sitio web en cuestión como cadena de texto:

```
>>> response.text
'\n\n\n\n\n\n<!DOCTYPE html>\n<html lang="en" dir="ltr">\n  <head>\n    <meta
↪charset="utf-8">\n    <meta http-equiv="X-UA-Compatible" content="IE=edge">\n
↪<meta name="viewport" content="width=device-width, initial-scale=1">\n\n    <meta
↪name="defaultLanguage" content="en">\n    <meta name="availableLanguages" content=
↪"en, es, fr, ja, pt_BR, uk, el, de, zh_Hans, zh_Hant, ru, he, eo">\n\n    \n\n
↪<title>PyPI · The Python Package Index</title>\n    <meta name="description"
↪content="The Python Package Index (PyPI) is a repository of software for the
↪Python programming language.">\n\n    <link rel="stylesheet" href="/static/css/
↪warehouse-ltr.69ee0d4e.css">\n    <link rel="stylesheet" href="/static/css/
↪fontawesome.6002a161.css">\n    <link rel="stylesheet" href="/static/css/regular.
↪98fbf39a.css">\n    <link rel="stylesheet" href="/static/css/solid.c3b5f0b5.css">\
↪n    <link rel="stylesheet" href="/static/css/brands.2c303be1.css">\n    <link rel=
↪"stylesheet" href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400,
↪400italic,600,600italic,700,700italic%7CSource+Code+Pro:500">\n    <noscript>\n
↪  <link rel="stylesheet" href="/static/css/noscript.d4ce1e76.css">\n'
```

**Nota:** Se ha recortado la salida a efectos visuales.

Algo que es realmente importante en una petición HTTP es comprobar el estado de la misma. Por regla general, si todo ha ido bien, deberíamos obtener un **código 200**, pero existen muchos otros códigos de estado de respuesta HTTP:

```
>>> response.status_code
200
```

**Truco:** Para evitar la comparación directa con el literal 200, existe la variable `requests.codes.ok`.

## 10.1.2 Tipos de peticiones

Con *requests* podemos realizar peticiones mediante cualquier método HTTP[2]. Para ello, simplemente usamos el método correspondiente del paquete:

[2] Métodos de petición HTTP.

| Método HTTP | Llamada |
|---|---|
| GET | `requests.get()` |
| POST | `requests.post()` |
| PUT | `requests.put()` |
| DELETE | `requests.delete()` |
| HEAD | `requests.head()` |
| OPTIONS | `requests.options()` |

### 10.1.3 Usando parámetros

Cuando se realiza una petición HTTP es posible incluir parámetros. Veamos distintas opciones que nos ofrece *requests* para ello.

#### Query string

En una petición GET podemos incluir parámetros en el llamado «query string». Los parámetros se definen mediante un *diccionario* con nombre y valor de parámetro.

Veamos un ejemplo sencillo. Supongamos que queremos **buscar paquetes de Python** que contengan la palabra «astro»:

```
>>> payload = {'q': 'astro'}

>>> response = requests.get('https://pypi.org', params=payload)

>>> response.url
'https://pypi.org/?q=astro'
```

**Truco:** El atributo `url` nos devuelve la URL a la se ha accedido. Útil en el caso de paso de parámetros.

#### Parámetros POST

Una petición POST, por lo general, siempre va acompañada de una serie de parámetros que típicamente podemos encontrar en un formulario web. Es posible realizar estas peticiones en *requests* adjuntando los parámetros que necesitemos en el mismo formato de diccionario que hemos visto para «query string».

Supongamos un ejemplo en el que tratamos de **logearnos en la página de GIPHY** con

nombre de usuario y contraseña. Para ello, lo primero que debemos hacer es inspeccionar[3] los elementos del formulario e identificar los nombres («name») de los campos. En este caso los campos son `email` y `password`:

```
>>> url = 'https://giphy.com/login'
>>> payload = {'email': 'sdelquin@gmail.com', 'password': '1234'}

>>> response = requests.post(url, data=payload)
>>> response.status_code
403
```

Hemos obtenido un código de estado 403 indicando que el acceso está prohibido.

### Envío de cabeceras

Hay veces que necesitamos modificar o añadir determinados campos en las cabeceras[4] de la petición. Su tratamiento también se realiza a base de diccionarios que son pasados al método correspondiente.

Uno de los usos más típicos de las cabeceras es el «user agent»[5] donde se especifica el tipo de navegador que realiza la petición. Supongamos un ejemplo en el que queremos especificar que **el navegador corresponde con un Google Chrome corriendo sobre Windows 10**:

```
>>> user_agent = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/97.0.4692.99 Safari/537.36'
>>> headers = {'user-agent': user_agent}

>>> response = requests.get('https://pypi.org', headers=headers)

>>> response.status_code
200
```

[3] Herramientas para desarrolladores en el navegador. Por ejemplo Chrome Dev Tools.

[4] Las cabeceras HTTP permiten al cliente y al servidor enviar información adicional junto a una petición o respuesta.

[5] El agente de usuario del navegador permite que el servidor identifique el sistema operativo y las características del navegador.

### 10.1.4 Analizando la respuesta

A continuación analizaremos distintos **elementos que forman parte de la respuesta HTTP** tras realizar la petición.

#### Contenido JSON

El formato JSON es ampliamente utilizado para el intercambio de datos entre aplicaciones. Hay ocasiones en las que la respuesta a una petición viene en dicho formato. Para facilitar su tratamiento *requests* nos proporciona un método que convierte el contenido JSON a un diccionario de Python.

Supongamos que queremos tener un **pronóstico del tiempo** en Santa Cruz de Tenerife. Existen múltiples servicios online que ofrecen datos meteorológicos. En este caso vamos a usar https://open-meteo.com/. La cuestión es que *los datos que devuelve esta API son en formato JSON*. Así que aprovecharemos para convertirlos de forma apropiada:

```
>>> sc_tfe = (28.4578025, -16.3563748)
>>> params = dict(latitude=sc_tfe[0], longitude=sc_tfe[1], hourly='temperature_2m')
>>> url = 'https://api.open-meteo.com/v1/forecast'

>>> response = requests.get(url, params=params)

>>> response.url
'https://api.open-meteo.com/v1/forecast?latitude=28.4578025&longitude=-16.3563748&
↪hourly=temperature_2m'

>>> data = response.json()

>>> type(data)
dict

>>> data.keys()
dict_keys(['utc_offset_seconds', 'elevation', 'latitude', 'hourly_units', 'longitude
↪', 'generationtime_ms', 'hourly'])
```

Ahora podríamos mostrar la predicción de temperatures de una manera algo más visual. Según la documentación de la API sabemos que la respuesta contiene 168 medidas de temperatura correspondientes a todas las horas durante 7 días. Supongamos que sólo queremos **mostrar la predicción de temperaturas hora a hora para el día de mañana**:

```
>>> temperatures = data['hourly']['temperature_2m']

>>> # Las temperaturas también incluyen el día de hoy
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> for i, temp in enumerate(temperatures[24:48], start=1):
...     print(f'{temp:4.1f}', end=' ')
...     if i % 6 == 0:
...         print()
...
12.0 11.9 11.9 11.8 11.8 11.7
11.7 11.7 11.6 12.0 12.8 13.6
13.9 14.0 14.1 13.9 13.7 13.3
12.8 12.2 11.8 11.7 11.6 11.5
```

### Cabeceras de respuesta

Tras una petición HTTP es posible recuperar las cabeceras que vienen en la respuesta a través del atributo `headers` como un diccionario:

```
>>> response = requests.get('https://pypi.org')
>>> response.status_code
200

>>> response.headers.get('Content-Type')
'text/html; charset=UTF-8'

>>> response.headers.get('Server')
'nginx/1.13.9'
```

### Cookies

Si una respuesta contiene «cookies»[6] es posible acceder a ellas mediante el diccionario `cookies`:

```
>>> response = requests.get('https://github.com')

>>> response.cookies.keys()
['_octo', 'logged_in', '_gh_sess']

>>> response.cookies.get('logged_in')
'no'
```

**Nota:** Las cookies también se pueden enviar en la petición usando `requests.get(url, cookies=cookies)`.

[6] Una cookie HTTP es una pequeña pieza de datos que un servidor envía al navegador web del usuario.

**Ejercicio**

Utilizando el paquete *requests*, haga una petición GET a https://twitter.com y obtenga los siguientes campos:

- Código de estado.
- Longitud de la respuesta.
- Valor de la cookie `guest_id`
- Valor de la cabecera `content-encoding`

## 10.1.5 Descargar un fichero

Hay ocasiones en las que usamos *requests* para descargar un fichero, bien sea en texto plano o binario. Veamos cómo proceder para cada tipo.

### Ficheros en texto plano

El procedimiento que utilizamos es descargar el contenido desde la url y *volcarlo a un fichero* de manera estándar:

```
>>> url = 'https://www.ine.es/jaxi/files/tpx/es/csv_bdsc/50155.csv'

>>> response = requests.get(url)

>>> response.status_code
200

>>> with open('data.csv', 'w') as f:
...     f.write(response.text)
...
```

**Consejo:** Usamos `response.text` para obtener el contenido ya que nos interesa en formato «unicode».

Podemos comprobar que el fichero se ha creado satisfactoriamente:

```
$ file data.csv
plain_text.csv: UTF-8 Unicode text, with CRLF line terminators
```

### Ficheros binarios

Para descargar ficheros binarios seguimos la misma estructura que para ficheros en texto plano, pero indicando el tipo binario a la hora de escribir en disco:

```
>>> url = 'https://www.ine.es/jaxi/files/tpx/es/xlsx/50155.xlsx'

>>> response = requests.get(url)

>>> response.status_code
200

>>> with open('data.xlsx', 'wb') as f:
...     f.write(response.content)
...
```

**Consejo:** Usamos `response.content` para obtener el contenido ya que nos interesa en formato «bytes».

Podemos comprobar que el fichero se ha creado satisfactoriamente:

```
$ file data.xlsx
data.xlsx: Microsoft OOXML
```

### Nombre de fichero

En los ejemplos anteriores hemos puesto el nombre de fichero «a mano». Pero podría darse la situación de necesitar el nombre de fichero que descargamos. Para ello existen dos aproximaciones en función de si aparece o no la clave «attachment» en las cabeceras de respuesta.

Podemos escribir la siguiente función para ello:

```
>>> def get_filename(response):
...     try:
...         return response.headers['Content-Disposition'].split(';')[1].split('=
↪')[1]
...     except (KeyError, IndexError):
...         return response.url.split('/')[-1]
...
```

Caso para el que no disponemos de la cabecera adecuada:

```
>>> url = 'https://media.readthedocs.org/pdf/pytest/latest/pytest.pdf'
>>> response = requests.get(url)
>>> 'attachment' in response.headers.get('Content-Disposition')
False

>>> get_filename(response)
'pytest.pdf'
```

Caso para el que sí disponemos de la cabecera adecuada:

```
>>> url = 'https://www.ine.es/jaxi/files/tpx/es/csv_bdsc/45070.csv'
>>> response = requests.get(url)
>>> 'attachment' in response.headers.get('Content-Disposition')
True

>>> get_filename(response)
'45070.csv'
```

## 10.2 beautifulsoup



El paquete Beautiful Soup es ampliamente utilizado en técnicas de «scraping» permitiendo

«parsear»[2] principalmente código HTML.[1]

```
$ pip install beautifulsoup4
```

### 10.2.1 Haciendo la sopa

Para empezar a trabajar con *Beautiful Soup* es necesario construir un objeto de tipo `BeautifulSoup` que reciba el contenido a «parsear»:

```
>>> from bs4 import BeautifulSoup

>>> contents = '''
... <html lang="en">
... <head>
...     <title>Just testing</title>
... </head>
... <body>
...     <h1>Just testing</h1>
...     <div class="block">
...       <h2>Some links</h2>
...       <p>Hi there!</p>
...       <ul id="data">
...         <li class="blue"><a href="https://example1.com">Example 1</a></li>
...         <li class="red"><a href="https://example2.com">Example 2</a></li>
...         <li class="gold"><a href="https://example3.com">Example 3</a></li>
...       </ul>
...     </div>
...     <div class="block">
...       <h2>Formulario</h2>
...       <form action="" method="post">
...         <label for="POST-name">Nombre:</label>
...         <input id="POST-name" type="text" name="name">
...         <input type="submit" value="Save">
...       </form>
...     </div>
...     <div class="footer">
...       This is the footer
...       <span class="inline"><p>This is span 1</p></span>
...       <span class="inline"><p>This is span 2</p></span>
...       <span class="inline"><p>This is span 2</p></span>
...     </div>
... </body>
```

(continué en la próxima página)

[2] Analizar y convertir una entrada en un formato interno que el entorno de ejecución pueda realmente manejar.

[1] Foto original de portada por Ella Olsson en Unsplash.

(proviene de la página anterior)

```
... </html>
... '''

>>> soup = BeautifulSoup(contents, features='html.parser')
```

**Atención:** Importar el paquete usando `bs4`. Suele llevar a equívoco por el nombre original.

A partir de aquí se abre un abanico de posibilidades que iremos desgranando en los próximos epígrafes.

## 10.2.2 Localizar elementos

Una de las tareas más habituales en técnicas de «scraping» y en «parsing» de contenido es la localización de determinadas elementos de interés.

### Fórmulas de localización

A continuación se muestran, mediante ejemplos, distintas fórmulas para localizar elementos dentro del DOM[3]:

- Localizar **todos los enlaces**:

```
>>> soup.find_all('a')
[<a href="https://example1.com">Example 1</a>,
 <a href="https://example2.com">Example 2</a>,
 <a href="https://example3.com">Example 3</a>]
```

  El primer *argumento posicional* de `find_all()` es el nombre del «tag» que queremos localizar.

- Localizar todos los **elementos con la clase `inline`**:

```
>>> soup.find_all(class_='inline')
[<span class="inline"><p>This is span 1</p></span>,
 <span class="inline"><p>This is span 2</p></span>,
 <span class="inline"><p>This is span 2</p></span>]
```

  Los *argumentos nominales* de `find_all()` se utilizan para localizar elementos que contengan el atributo referenciado.

[3] Document Object Model en español Modelo de Objetos del Documento.

**Truco:** Si el atributo a localizar tiene guiones medios (por ejemplo `aria-label`) no podremos usarlo como nombre de argumento (error sintáctico). Pero sí podemos usar un diccionario en su lugar:

```
soup.find_all(attrs={'aria-label': 'box'})
```

- Localizar todos los «divs» con la clase `footer`:

```
>>> soup.find_all('div', class_='footer')  # □ soup.find_all('div', 'footer')
[<div class="footer">
       This is the footer
       <span class="inline"><p>This is span 1</p></span>
 <span class="inline"><p>This is span 2</p></span>
 <span class="inline"><p>This is span 2</p></span>
 </div>]
```

- Localizar todos los elementos cuyo atributo `type` tenga el valor `text`:

```
>>> soup.find_all(type='text')
[<input id="POST-name" name="name" type="text"/>]
```

- Localizar todos los los `h2` que contengan el texto `Formulario`:

```
>>> soup.find_all('h2', string='Formulario')
[<h2>Formulario</h2>]
```

- Localizar todos los elementos de título `h1`, `h2`, `h3`, `....` Esto lo podemos atacar usando *expresiones regulares*:

```
>>> soup.find_all(re.compile(r'^h\d+.*'))
[<h1>Just testing</h1>, <h2>Some links</h2>, <h2>Formulario</h2>]
```

- Localizar todos los «input» y todos los «span»:

```
>>> soup.find_all(['input', 'span'])
[<input id="POST-name" name="name" type="text"/>,
 <input type="submit" value="Save"/>,
 <span class="inline"><p>This is span 1</p></span>,
 <span class="inline"><p>This is span 2</p></span>,
 <span class="inline"><p>This is span 2</p></span>]
```

- Localizar todos los párrafos que están dentro del pie de página (usando **selectores CSS**):

```
>>> soup.select('.footer p')
[<p>This is span 1</p>, <p>This is span 2</p>, <p>This is span 2</p>]
```

**Nota:** En este caso se usa el método `select()`.

### Localizar único elemento

Hasta ahora hemos visto las funciones `find_all()` y `select()` que localizan un conjunto de elementos. Incluso en el caso de encontrar sólo un elemento, se devuelve una lista con ese único elemento.

*Beautiful Soup* nos proporciona la función `find()` que trata de **localizar un único elemento**. Hay que tener en cuenta dos circunstancias:

- En caso de que el elemento buscado no exista, se devuelve *None*.
- En caso de que existan múltiples elementos, se devuelve el primero.

Veamos algunos ejemplos de esto:

```
>>> soup.find('form')
<form action="" method="post">
<label for="POST-name">Nombre:</label>
<input id="POST-name" name="name" type="text"/>
<input type="submit" value="Save"/>
</form>

>>> # Elemento que no existe
>>> soup.find('strange-tag')
>>>

>>> # Múltiples "li". Sólo se devuelve el primero
>>> soup.find('li')
<li class="blue"><a href="https://example1.com">Example 1</a></li>
```

### Localizar desde elemento

Todas las búsquedas se pueden realizar desde cualquier elemento preexistente, no únicamente desde la raíz del DOM.

Veamos un ejemplo de ello. Si tratamos de **localizar todos los títulos «h2»** vamos a encontrar dos de ellos:

```
>>> soup.find_all('h2')
[<h2>Some links</h2>, <h2>Formulario</h2>]
```

Pero si, previamente, nos ubicamos en el segundo bloque de contenido, sólo vamos a encontrar uno de ellos:

```
>>> second_block = soup.find_all('div', 'block')[1]

>>> second_block
<div class="block">
<h2>Formulario</h2>
<form action="" method="post">
<label for="POST-name">Nombre:</label>
<input id="POST-name" name="name" type="text"/>
<input type="submit" value="Save"/>
</form>
</div>

>>> second_block.find_all('h2')
[<h2>Formulario</h2>]
```

### Otras funciones de búsqueda

Hay definidas una serie de funciones adicionales de búsqueda para cuestiones más particulares:

- Localizar los **«div» superiores** a partir de un elemento concreto:

  ```
  >>> gold = soup.find('li', 'gold')

  >>> gold.find_parents('div')
  [<div class="block">
   <h2>Some links</h2>
   <p>Hi there!</p>
   <ul id="data">
   <li class="blue"><a href="https://example1.com">Example 1</a></li>
   <li class="red"><a href="https://example2.com">Example 2</a></li>
   <li class="gold"><a href="https://example3.com">Example 3</a></li>
   </ul>
   </div>]
  ```

  Se podría decir que la función `find_all()` busca en *descendientes* y que la función `find_parents()` busca en *ascendientes*.

  También existe la versión de esta *función que devuelve un único elemento*: `find_parent()`.

- Localizar los **elementos hermanos siguientes** a uno dado:

```
>>> blue_li = soup.find('li', 'blue')

>>> blue_li.find_next_siblings()
[<li class="red"><a href="https://example2.com">Example 2</a></li>,
<li class="gold"><a href="https://example3.com">Example 3</a></li>]
```

  Al igual que en las anteriores, es posible aplicar un filtro al usar esta función.

  También existe la versión de esta *función que devuelve un único elemento*: `find_next_sibling()`.

- Localizar los **elementos hermanos anteriores** a uno dado:

```
>>> gold_li = soup.find('li', 'gold')

>>> gold_li.find_previous_siblings()
[<li class="red"><a href="https://example2.com">Example 2</a></li>,
<li class="blue"><a href="https://example1.com">Example 1</a></li>]
```

  Al igual que en las anteriores, es posible aplicar un filtro al usar esta función.

  También existe la versión de esta *función que devuelve un único elemento*: `find_previous_sibling()`.

- Localizar **todos los elementos a continuación** de uno dado:

```
>>> submit = soup.find('input', type='submit')

>>> submit.find_all_next()
[<div class="footer">
 This is the footer
 <span class="inline"><p>This is span 1</p></span>
 <span class="inline"><p>This is span 2</p></span>
 <span class="inline"><p>This is span 2</p></span>
 </div>,
 <span class="inline"><p>This is span 1</p></span>,
 <p>This is span 1</p>,
 <span class="inline"><p>This is span 2</p></span>,
 <p>This is span 2</p>,
 <span class="inline"><p>This is span 2</p></span>,
 <p>This is span 2</p>]
```

  Al igual que en las anteriores, es posible aplicar un filtro al usar esta función.

  También existe la versión de esta *función que devuelve un único elemento*: `find_next()`.

- Localizar **todos los elementos previos** a uno dado:

```
>>> ul_data = soup.find('ul', id='data')

>>> ul_data.find_all_previous(['h1', 'h2'])
[<h2>Some links</h2>, <h1>Just testing</h1>]
```

También existe la versión de esta *función que devuelve un único elemento*: `find_previous()`.

Si hubiéramos hecho esta búsqueda usando `find_parents()` no habríamos obtenido el mismo resultado ya que los elementos de título no son elementos superiores de «ul»:

```
>>> ul_data.find_parents(['h1', 'h2'])
[]
```

### Atajo para búsquedas

Dado que la función `find_all()` es la más utilizada en *Beautiful Soup* se ha implementado un atajo para llamarla:

```
>>> soup.find_all('span')
[<span class="inline"><p>This is span 1</p></span>,
 <span class="inline"><p>This is span 2</p></span>,
 <span class="inline"><p>This is span 2</p></span>]

>>> soup('span')
[<span class="inline"><p>This is span 1</p></span>,
 <span class="inline"><p>This is span 2</p></span>,
 <span class="inline"><p>This is span 2</p></span>]
```

Aunque uno de los preceptos del *Zen de Python* es «Explicit is better than implicit», el uso de estos atajos puede estar justificado en función de muchas circunstancias.

## 10.2.3 Acceder al contenido

Simplificando, podríamos decir que cada elemento de la famosa «sopa» de *Beautiful Soup* puede ser un `bs4.element.Tag` o un «string».

Para el caso de los «tags» existe la posibilidad de acceder a su contenido, al nombre del elemento o a sus atributos.

### Nombre de etiqueta

Podemos conocer el nombre de la etiqueta de un elemento usando el atributo `name`:

```
>>> soup.name
'[document]'

>>> elem = soup.find('ul', id='data')
>>> elem.name
'ul'

>>> elem = soup.find('h1')
>>> elem.name
'h1'
```

**Truco:** Es posible modificar el nombre de una etiqueta con una simple asignación.

### Acceso a atributos

Los atributos de un elemento están disponibles como claves de un diccionario:

```
>>> elem = soup.find('input', id='POST-name')

>>> elem
<input id="POST-name" name="name" type="text"/>

>>> elem['id']
'POST-name'

>>> elem['name']
'name'

>>> elem['type']
'text'
```

Exite una forma de acceder al diccionario completo de atributos:

```
>>> elem.attrs
{'id': 'POST-name', 'type': 'text', 'name': 'name'}
```

**Truco:** Es posible modificar el valor de un atributo con una simple asignación.

### Contenido textual

Es necesario aclarar las distintas opciones que proporciona *Beautiful Soup* para acceder al contenido textual de los elementos del DOM.

| Atributo | Devuelve |
|---|---|
| `text` | Una cadena de texto con todos los contenidos textuales del elemento incluyendo espacios y saltos de línea |
| `strings` | Un generador de todos los contenidos textuales del elemento incluyendo espacios y saltos de línea |
| `stripped_strings` | Un generador de todos los contenidos textuales del elemento eliminando espacios y saltos de línea |
| `string` | Una cadena de texto con el contenido del elemento, siempre que contenga un único elemento textual |

Ejemplos:

```
>>> footer = soup.find(class_='footer')

>>> footer.text
'\n      This is the footer\n      This is span 1\nThis is span 2\nThis is span 2\n'

>>> list(footer.strings)
['\n      This is the footer\n      ',
 'This is span 1',
 '\n',
 'This is span 2',
 '\n',
 'This is span 2',
 '\n']

>>> list(footer.stripped_strings)
['This is the footer', 'This is span 1', 'This is span 2', 'This is span 2']

>>> footer.string       # El "footer" contiene varios elementos

>>> footer.span.string  # El "span" sólo contiene un elemento
'This is span 1'
```

### Mostrando elementos

Cualquier elemento del DOM que seleccionemos mediante este paquete se representa con el código HTML que contiene. Por ejemplo:

```
>>> data = soup.find(id='data')

>>> data
<ul id="data">
<li class="blue"><a href="https://example1.com">Example 1</a></li>
<li class="red"><a href="https://example2.com">Example 2</a></li>
<li class="gold"><a href="https://example3.com">Example 3</a></li>
</ul>
```

Existe la posibilidad de mostrar el código HTML en formato «mejorado» a través de la función `prettify()`:

```
>>> pretty_data = data.prettify()

>>> print(pretty_data)
<ul id="data">
 <li class="blue">
  <a href="https://example1.com">
   Example 1
  </a>
 </li>
 <li class="red">
  <a href="https://example2.com">
   Example 2
  </a>
 </li>
 <li class="gold">
  <a href="https://example3.com">
   Example 3
  </a>
 </li>
</ul>
```

### 10.2.4 Navegar por el DOM

Además de localizar elementos, este paquete permite moverse por los elementos del DOM de manera muy sencilla.

#### Moverse hacia abajo

Para ir profundizando en el DOM podemos utilizar los **nombres de los «tags» como atributos del objeto**, teniendo en cuenta que si existen múltiples elementos sólo se accederá al primero de ellos:

```
>>> soup.div.p
<p>Hi there!</p>

>>> soup.form.label
<label for="POST-name">Nombre:</label>

>>> type(soup.span)
bs4.element.Tag
```

Existe la opción de obtener el **contenido (como lista) de un determinado elemento**:

```
>>> type(soup.form)  # todos los elementos del DOM son de este tipo
bs4.element.Tag

>>> soup.form.contents
['\n',
 <label for="POST-name">Nombre:</label>,
 '\n',
 <input id="POST-name" name="name" type="text"/>,
 '\n',
 <input type="submit" value="Save"/>,
 '\n']

>>> type(soup.form.contents)
list
```

**Advertencia:** En la lista que devuelve `contents` hay mezcla de «strings» y objetos `bs4.element.Tag`.

Si no se quiere explicitar el contenido de un elemento como lista, también es posible usar un *generador* para **acceder al mismo de forma secuencial**:

```
>>> soup.form.children
<list_iterator at 0x106643100>

>>> for elem in soup.form.children:
...     print(repr(elem))
...
'\n'
<label for="POST-name">Nombre:</label>
'\n'
<input id="POST-name" name="name" type="text"/>
'\n'
<input type="submit" value="Save"/>
'\n'
```

El atributo `contents` sólo tiene en cuenta descendientes directos. Si queremos **acceder a cualquier elemento descendiente (de manera recursiva)** tenemos que usar `descendants`:

```
>>> block = soup.find_all('div')[1]

>>> block.contents
['\n',
 <h2>Formulario</h2>,
 '\n',
 <form action="" method="post">
 <label for="POST-name">Nombre:</label>
 <input id="POST-name" name="name" type="text"/>
 <input type="submit" value="Save"/>
 </form>,
 '\n']

>>> block.descendants
<generator object Tag.descendants at 0x10675d200>

>>> list(block.descendants)
['\n',
 <h2>Formulario</h2>,
 'Formulario',
 '\n',
 <form action="" method="post">
 <label for="POST-name">Nombre:</label>
 <input id="POST-name" name="name" type="text"/>
 <input type="submit" value="Save"/>
 </form>,
 '\n',
 <label for="POST-name">Nombre:</label>,
```

(continué en la próxima página)

(proviene de la página anterior)

```
 'Nombre:',
 '\n',
 <input id="POST-name" name="name" type="text"/>,
 '\n',
 <input type="submit" value="Save"/>,
 '\n',
 '\n']
```

**Importante:** Tener en cuenta que `descendants` es un generador que devuelve un iterable.

### Moverse hacia arriba

Para acceder al **elemento superior de otro dado**, podemos usar el atributo `parent`:

```
>>> li = soup.find('li', 'blue')

>>> li.parent
<ul id="data">
<li class="blue"><a href="https://example1.com">Example 1</a></li>
<li class="red"><a href="https://example2.com">Example 2</a></li>
<li class="gold"><a href="https://example3.com">Example 3</a></li>
</ul>
```

También podemos acceder a **todos los elementos superiores (ascendientes)** usando el generador `parents`:

```
>>> for elem in li.parents:
...     print(elem.name)
...
ul
div
body
html
[document]
```

### Otros movimientos

En la siguiente tabla se recogen el resto de atributos que nos permiten movernos a partir de un elemento del DOM:

| Atributo | Descripción |
|---|---|
| `next_sibling` | Obtiene el siguiente elemento «hermano» |
| `previous_sibling` | Obtiene el anterior elemento «hermano» |
| `next_siblings` | Obtiene los siguientes elementos «hermanos» (iterador) |
| `previous_siblings` | Obtiene los anteriores elementos «hermanos» (iterador) |
| `next_element` | Obtiene el siguiente elemento |
| `previous_element` | Obtiene el anterior elemento |

**Advertencia:** Con estos accesos también se devuelven los saltos de línea `'\n'` como elementos válidos. Si se quieren evitar, es preferible usar las *funciones definidas aquí*.

---

**Ejercicio**

Escriba un programa en Python que obtenga de https://pypi.org datos estructurados de los «Trending projects» y los muestre por pantalla utilizando el siguiente formato:

```
<nombre-del-paquete>,<versión>,<descripción>,<url>
```

Se recomienda usar el paquete *requests* para obtener el código fuente de la página. Hay que tener en cuenta que el listado de paquetes cambia cada pocos segundos, a efectos de comprobación.

---

# 10.3 selenium



Selenium es un proyecto que permite **automatizar navegadores**. Está principalmente enfocado al testeo de aplicaciones web pero también permite desarrollar potentes flujos de trabajo como es el caso de las técnicas de *scraping*.[1]

Existen múltiples «bindings»[2] pero el que nos ocupa en este caso es el de Python:

```
$ pip install selenium
```

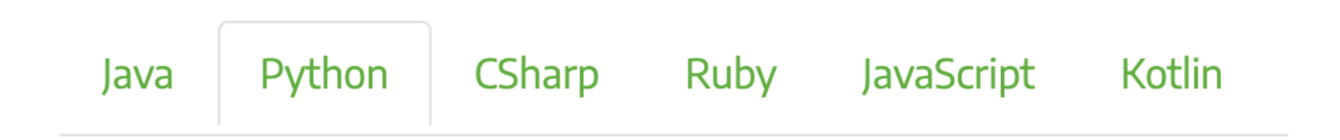
## 10.3.1 Pasos previos

### Documentación

Recomendamos la documentación oficial de Selenium como punto de entrada a la librería. Eso sí, como ya hemos comentado previamente, existen adaptaciones para Python, Java, CSharp, Ruby, JavaScript y Kotlin, por lo que es conveniente fijar la pestaña de **Python** en los ejemplos de código:

Es igualmente importante manejar la documentación de la API para Python.

[1] Foto original de portada por Andy Kelly en Unsplash.
[2] Adaptación (interface) de la herramienta a un lenguaje de programación concreto.




#### Prerequisitos

##### Navegador web

Selenium necesita un **navegador web** instalado en el sistema para poder funcionar. Dentro de las opciones disponibles están Chrome, Firefox, Edge, Internet Explorer y Safari. En el caso de este documento vamos a utilizar Firefox. Su descarga e instalación es muy sencilla.

##### Driver

Además de esto, también es necesario disponer un **«webdriver»** que permita manejar el navegador (a modo de marioneta). Cada navegador tiene asociado un tipo de «driver». En el caso de Firefox hablamos de geckodriver. Su descarga e instalación es muy sencilla.

## 10.3.2 Configuración del driver

El «driver» es el manejador de las peticiones del usuario. Se trata del objeto fundamental en Selenium que nos permitirá interactuar con el navegador y los sitios web.

#### Inicialización del driver

Para inicializar el «driver», en su versión más simple, usaremos el siguiente código:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
```

En este momento se abrirá una ventana con el navegador Firefox.

**Truco:** Es posible usar otros navegadores. La elección de este documento por Firefox tiene que ver con cuestiones de uso durante los últimos años.

### Capacidades del navegador

Cuando inicializamos el «driver» podemos asignar ciertas «capacidades» al navegador. Las podemos dividir en dos secciones: opciones y perfil.

#### Opciones del navegador

Una de las opciones más utilizadas es la capacidad de ocultar la ventana del navegador. Esto es útil cuando ya hemos probado que todo funciona y queremos automatizar la tarea:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.firefox.options import Options

>>> options = Options()
>>> options.headless = True

>>> driver = webdriver.Firefox(options=options)
```

El resto de opciones se pueden consultar en la documentación de la API[3].

#### Perfil del navegador

Es posible definir un perfil personalizado para usarlo en el navegador controlado por el «driver».

Como ejemplo, podríamos querer desactivar javascript en el navegador (por defecto está activado). Esto lo haríamos de la siguiente manera:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.firefox.options import Options
>>> from selenium.webdriver.firefox.firefox_profile import FirefoxProfile

>>> firefox_profile = FirefoxProfile()
>>> firefox_profile.set_preference('javascript.enabled', False)

>>> options=Options()
>>> options.profile = firefox_profile

>>> driver = webdriver.Firefox(options=options)
```

Existe una cantidad ingente de parámetros configurables en el perfil de usuario[3]. Se pueden consultar en estos dos enlaces:

- https://searchfox.org/mozilla-release/source/modules/libpref/init/all.js

[3] En este caso aplicable al navegador Firefox.

- https://searchfox.org/mozilla-release/source/browser/app/profile/firefox.js

### Fichero de log

Desde la primera vez que inicializamos el «driver», se crea un fichero de log en el directorio de trabajo con el nombre `geckodriver.log`[3]. En este fichero se registran todos los mensajes que se producen durante el tiempo de vida del navegador.

Es posible, aunque no recomendable, **evitar que se cree el fichero de log** de la siguiente manera:

```
>>> import os

>>> driver = webdriver.Firefox(options=options, service_log_path=os.devnull)
```

**Nota:** De igual modo, con el método anterior podemos cambiar la ruta y el nombre del fichero de log.

## 10.3.3 Navegando

La forma de **acceder a una url** es utilizar el método `.get()`:

```
>>> driver.get('https://aprendepython.es')
```

**Importante:** Cuando se navega a un sitio web, Selenium espera (por defecto) a que la propiedad `document.readyState` tenga el valor `complete`. Esto no implica necesariamente que la página se haya cargado completamente, especialmente en páginas que usan mucho javascript para cargar contenido dinámicamente.

Podemos movernos **«hacia adelante» y «hacia detrás»** con:

```
>>> driver.forward()
>>> driver.back()
```

Si fuera necesario, también existe la posibilidad de **refrescar la página**:

```
>>> driver.refresh()
```

Una vez terminadas todas las operaciones requeridas, es altamente recomendable **salir del navegador** para liberar los recursos que pueda estar utilizando:

```
>>> driver.quit()
```

### 10.3.4 Localizando elementos

Una vez que hemos accedido a un sitio web, estamos en disposición de localizar elementos dentro del DOM[4]. El objeto `driver` nos ofrece las siguientes funciones para ello:

| Acceso | Función | Localizador |
|---|---|---|
| Clase | `find_elements_by_class_name()` | `By.CLASS_NAME` |
| Selector CSS | `find_elements_by_css_selector()` | `By.CSS_SELECTOR` |
| Atributo ID | `find_elements_by_id()` | `By.ID` |
| Texto en enlace | `find_elements_by_link_text()` | `By.LINK_TEXT` |
| Texto en enlace (parcial) | `find_elements_by_partial_link_text()` | `By.PARTIAL_LINK_TEXT` |
| Atributo NAME | `find_elements_by_name()` | `By.NAME` |
| Nombre de etiqueta | `find_elements_by_tag_name()` | `By.TAG_NAME` |
| XPath | `find_elements_by_xpath()` | `By.XPATH` |

Todas estas funciones tienen su correspondiente versión **para devolver un único elemento** que cumpla con el filtro especificado. En caso de que hayan varios, sólo se devolverá el primero de ellos. El nombre de estas funciones sigue el patrón en singular:

```
find_element_by_<accesor>()
```

Si te estás preguntando **para qué sirve el localizador** de la tabla anterior, es porque también existe la opción de localizar elementos mediante una función genérica:

```
>>> from selenium.webdriver.common.by import By

>>> # Estas dos llamadas tienen el mismo significado
>>> driver.find_elements_by_class_name('matraca')
>>> driver.find_elements(By.CLASS_NAME, 'matraca')
```

Veamos un ejemplo práctico de esto. Supongamos que queremos **obtener los epígrafes principales de la tabla de contenidos de «Aprende Python»**:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
>>> driver.get('https://aprendepython.es')
```

(continué en la próxima página)

[4] Document Object Model.

(proviene de la página anterior)

```
>>> toc = driver.find_elements_by_css_selector('div.sidebar-tree li.toctree-l1')

>>> for heading in toc:
...     print(heading.text)
...
Introducción
Entornos de desarrollo
Tipos de datos
Control de flujo
Estructuras de datos
Modularidad
Procesamiento de texto
Ciencia de datos
Scraping
```

**Truco:** Un poco más adelante veremos la forma de acceder a la información que contiene cada elemento del DOM.

Cada elemento que obtenemos con las funciones de localización es de tipo FirefoxWebElement:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
>>> driver.get('https://aprendepython.es')

>>> element = driver.find_element_by_id('aprende-python')

>>> type(element)
selenium.webdriver.firefox.webelement.FirefoxWebElement
```

### 10.3.5 Interacciones

Si bien el acceso a la información de un sitio web puede ser un objetivo en sí mismo, para ello podríamos usar herramientas como *requests*. Sin embargo, cuando entra en juego la interacción con los elementos del DOM, necesitamos otro tipo de aproximaciones.

Selenium nos permite hacer clic en el lugar deseado, enviar texto por teclado, borrar una caja de entrada o manejar elementos de selección, entre otros.

### Clic

Para **hacer clic** utilizamos la función homónima. Veamos un ejemplo en el que accedemos a https://amazon.es y tenemos que aceptar las «cookies» haciendo clic en el botón correspondiente:

```
>>> driver = webdriver.Firefox()

>>> driver.get('https://amazon.es')

>>> accept_cookies = driver.find_element_by_id('sp-cc-accept')

>>> accept_cookies.click()
```

### Inspeccionando el DOM

Una tarea inherente a las técnicas de «scraping» y a la automatización de comportamientos para navegadores web es la de **inspeccionar los elementos del DOM**. Esto se puede hacer desde las herramientas para desarrolladores que incluyen los navegadores[5].

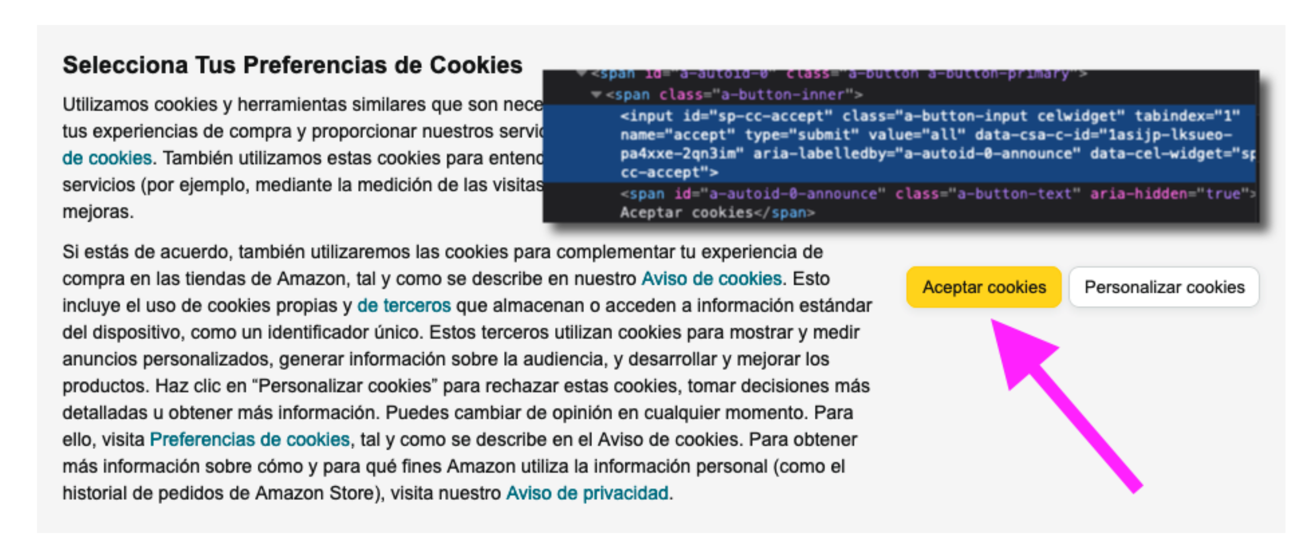



Figura 1: Botón de «cookies» en amazon.es

Para el ejemplo anterior de Amazon en el que debemos identificar el botón para aceptar «cookies», abrimos el inspector de Firefox y descubrimos que su `id` es `sp-cc-accept`. Si no lo tuviéramos disponible habría que hacer uso de otros localizadores como «xpath» o selectores de estilo.

---

**Ejercicio**

[5] Para Firefox tenemos a disposición la herramienta Inspector.

Escriba un programa en Python que, utilizando Selenium, pulse el botón de «¡JUGAR!» en el sitio web https://wordle.danielfrg.com/. Los selectores «xpath» pueden ser de mucha ayuda.

### Enviar texto

Típicamente encontraremos situaciones donde habrá que enviar texto a algún campo de entrada de un sitio web. Selenium nos permite hacer esto.

Veamos un ejemplo en el que tratamos de **hacer login sobre PyPI**:

```
>>> driver = webdriver.Firefox()

>>> driver.get('https://pypi.org/account/login/')

>>> username = driver.find_element_by_id('username')
>>> password = driver.find_element_by_id('password')

>>> username.send_keys('sdelquin')
>>> password.send_keys('1234')

>>> login_btn_xpath = '//*[@id="content"]/div/div/form/div[2]/div[3]/div/div/input'
>>> login_btn = driver.find_element_by_xpath(login_btn_xpath)

>>> login_btn.click()
```

En el caso de que queramos enviar alguna tecla «especial», Selenium nos proporciona un conjunto de símbolos para ello, definidos en selenium.webdriver.common.keys.

Por ejemplo, si quisiéramos **enviar las teclas de cursor**, haríamos lo siguiente:

```
>>> from selenium.webdriver.common.keys import Keys

>>> element.send_keys(Keys.RIGHT)  # →
>>> element.send_keys(Keys.DOWN)   # ↓
>>> element.send_keys(Keys.LEFT)   # ←
>>> element.send_keys(Keys.UP)     # ↑
```

**Ejercicio**

Escriba un programa en Python utilizando Selenium que, dada una palabra de 5 caracteres, permita enviar ese «string» a https://wordle.danielfrg.com/ para jugar.

Tenga en cuenta lo siguiente:

- En primer lugar hay que pulsar el botón de «¡JUGAR!».

- El elemento sobre el que enviar texto podría ser directamente el «body».
- Puede ser visualmente interesante poner un `time.sleep(0.5)` tras la inserción de cada letra.
- Una vez enviada la cadena de texto hay que pulsar ENTER.

### Borrar contenido

Si queremos borrar el contenido de un elemento web editable, típicamente una caja de texto, lo podemos hacer usando el método `.clear()`.

### Manejo de selects

Los elementos de selección `<select>` pueden ser complicados de manejar a nivel de automatización. Para suplir esta dificultad, Selenium proporciona el objecto `Select`.

Supongamos un ejemplo en el que modificamos el idioma de búsqueda de Wikipedia. En primer lugar hay que acceder a la web y seleccionar el desplegable del idioma de búsqueda:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.support.select import Select

>>> driver = webdriver.Firefox()
>>> driver.get('https://wikipedia.org')

>>> lang = driver.find_element_by_id('searchLanguage')
>>> lang_handler = Select(lang)
```

Ahora vamos a seleccionar el *idioma inglés* como idioma de búsqueda:

```
>>> # Selección por el índice que ocupa cada "option" (base 0)
>>> lang_handler.select_by_index(17)

>>> # Selección por el campo "value" de cada "option"
>>> lang_handler.select_by_value('en')

>>> # Selección por el texto visible de cada "option"
>>> lang_handler.select_by_visible_text('English')
```

**Truco:** Estas tres funciones tienen su correspondiente `deselect_by_<accesor>`.

Si queremos identificar las opciones que están actualmente seleccionadas, podemos usar los siguientes atributos:

```
>>> lang_handler.all_selected_options
[<selenium.webdriver.firefox.webelement.FirefoxWebElement (session="8612e5b7-6e66-
↪4121-8869-ffce4139d197", element="8433ffdb-a8ad-4e0e-9367-d63fe1418b94")>]

>>> lang_handler.first_selected_option
<selenium.webdriver.firefox.webelement.FirefoxWebElement (session="8612e5b7-6e66-
↪4121-8869-ffce4139d197", element="8433ffdb-a8ad-4e0e-9367-d63fe1418b94")>
```

También es posible listar todas las opciones disponibles en el elemento `select`:

```
>>> lang_handler.options
[..., ..., ...]
```

**Ver también:**

API del objeto Select

### 10.3.6 Acceso a atributos

Como ya hemos comentado, los objetos del DOM con los que trabaja Selenium son de tipo `FirefoxWebElement`. Veremos los mecanismos disponibles para poder acceder a sus atributos.

Para ejemplificar el acceso a la información de los elementos del DOM, vamos a **localizar el botón de descarga** en la web de Ubuntu:

```
>>> from selenium import webdriver

>>> driver = webdriver.Firefox()
>>> driver.get('https://ubuntu.com/')

>>> # Aceptar las cookies
>>> cookies = driver.find_element_by_id('cookie-policy-button-accept')
>>> cookies.click()

>>> download_btn = driver.find_element_by_id('takeover-primary-url')
```

#### Nombre de etiqueta

```
>>> download_btn.tag_name
'a'
```

### Tamaño y posición

Para cada elemento podemos obtener un diccionario que contiene la posición en pantalla $(x, y)$ acompañado del ancho y del alto (todo en píxeles):

```
>>> download_btn.rect
{'x': 120.0,
 'y': 442.3999938964844,
 'width': 143.64999389648438,
 'height': 36.80000305175781}
```

### Estado

Veamos las funciones disponibles para saber si un elemento se está mostrando, está habilitado o está seleccionado:

```
>>> download_btn.is_displayed()
True

>>> download_btn.is_enabled()
True

>>> download_btn.is_selected()
False
```

### Propiedad CSS

En caso de querer conocer el valor de cualquier propiedad CSS de un determinado elemento, lo podemos conseguir así:

```
>>> download_btn.value_of_css_property('background-color')
'rgb(14, 132, 32)'

>>> download_btn.value_of_css_property('font-size')
'16px'
```

### Texto del elemento

Cuando un elemento incluye texto en su contenido, ya sea de manera directa o mediante elementos anidados, es posible acceder a esta información usando:

```
>>> download_btn.text
'Download Now'
```

### Elemento superior

Selenium también permite obtener el elemento superior que contiene a otro elemento dado:

```
>>> download_btn.parent
<selenium.webdriver.firefox.webdriver.WebDriver (session="8612e5b7-6e66-4121-8869-
↪ffce4139d197")>
```

### Propiedad de elemento

De manera más genérica, podemos obtener el valor de cualquier atributo de un elemento del DOM a través de la siguiente función:

```
>>> download_btn.get_attribute('href')
'https://ubuntu.com/engage/developer-desktop-productivity-whitepaper
```

## 10.3.7 Esperas

Cuando navegamos a un sitio web utilizando `driver.get()` es posible que el elemento que estamos buscando no esté aún cargado en el DOM porque existan peticiones asíncronas pendientes o contenido dinámico javascript. Es por ello que Selenium pone a nuestra disposición una serie de **esperas explícitas** hasta que se cumpla una determinada condición.

Las esperas explícitas suelen hacer uso de condiciones de espera. Cada una de estas funciones se puede utilizar para un propósito específico. Quizás una de las funciones más habituales sea `presence_of_element_located()`.

Veamos un ejemplo en el que cargamos la web de Stack Overflow y esperamos a que el pie de página esté disponible:

```
>>> from selenium.webdriver.support.ui import WebDriverWait
>>> from selenium.webdriver.support import expected_conditions as EC
>>> from selenium.webdriver.common.by import By

>>> driver.get('https://stackoverflow.com')
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> footer = WebDriverWait(driver, 10).until(
...     EC.presence_of_element_located((By.ID, 'footer')))

>>> print(footer.text)
STACK OVERFLOW
Questions
Jobs
Developer Jobs Directory
Salary Calculator
Help
Mobile
...
```

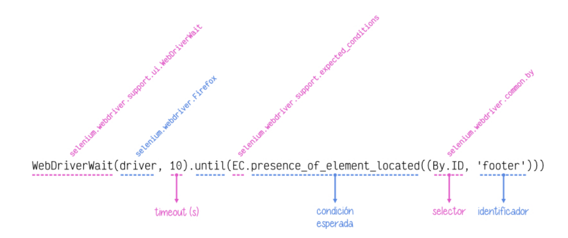



Figura 2: Anatomía de una condición de espera

**Atención:** En el caso de que el elemento por el que estamos esperando no «aparezca» en el DOM, y una vez pasado el tiempo de «timeout», Selenium eleva una *excepción* de tipo `selenium.common.exceptions.TimeoutException`.

## 10.3.8 Ejecutar javascript

Puede llegar a ser muy útil la ejecución de javascript en el navegador. La casuística es muy variada. En cualquier caso, Selenium nos proporciona el método `execute_script()` para esta tarea.

Supongamos un ejemplo en el que queremos **navegar a la web de GitHub y hacer «scroll» hasta el final de la página**:

```
>>> driver = webdriver.Firefox()
>>> driver.get('https://github.com')

>>> body = driver.find_element_by_tag_name('body')

>>> driver.execute_script('arguments[0].scrollIntoView(false)', body)
```

Cuando en la función `execute_script()` se hace referencia al array `arguments[]` podemos pasar elementos Selenium como argumentos y aprovechar así las potencialidades javascript. El primer argumento corresponde al índice 0, el segundo argumento al índice 1, y así sucesivamente.

---

**Ejercicio**

Escriba un programa en Python que permita sacar un listado de supermercados Mercadona dada una geolocalización `(lat,lon)` como dato de entrada.

Pasos a seguir:

1. Utilizar el siguiente *f-string* para obtener la url de acceso: `f'https://info.mercadona.es/es/supermercados?coord={lat}%2C{lon}'`
2. Aceptar las cookies al acceder al sitio web.
3. Hacer scroll hasta el final de la página para hacer visible el botón «Ver todos». Se recomienda usar javascript para ello.
4. Localizar el botón «Ver todos» y hacer clic para mostrar todos los establecimientos (de la geolocalización). Se recomienda una espera explícita con acceso por «xpath».
5. Recorrer los elementos desplegados `li` y mostrar el contenido textual de los elementos `h3` que hay en su interior.

Como detalle final, y una vez que compruebe que el programa funciona correctamente, aproveche para inicializar el «driver» *ocultando la ventana del navegador*.

Puede probar su programa con la localización de Las Palmas de Gran Canaria (28.1035677, -15.5319742).

---